# Open System Services System Calls Reference Manual


**Abstract**

This manual documents part of the HP NonStop Open System Services (OSS) application program interface. It is written for system and application programmers.


**Product Version**

N.A.

**Supported Release Version Updates (RVUs)**

This manual supports J06.03 and all subsequent J-series RVUs, H06.08 and all subsequent H-series RVUs, and G06.29 and all subsequent G-series RVUs until otherwise indicated by its replacement publication.


| Part Number | Published |
|---|---|
| 527186-023 | February 2014 |

## Document History

| Part Number | Product Version | Published |
|---|---|---|
| 527186-019 | N/A | February 2012 |
| 527186-020 | N/A | August 2012 |
| 527186-021 | N/A | February 2013 |
| 527186-022 | N/A | August 2013 |
| 527186-023 | N/A | February 2014 |

# Contents

## LIST OF TABLES

# What Is New in This Manual

This section describes changes made to the *Open System Services System Calls Reference Manual* since the last edition (527186-022).

Unless otherwise indicated in the text, discussions of native mode behavior, processes, and so forth apply to both the TNS/R code that runs on systems running G-series RVUs and to the TNS/E code that runs on systems running J-series RVUs or H-series RVUs. Discussions of TNS or accelerated code behavior in the OSS environment apply only to systems running G-series RVUs; systems running J-series RVUs or H-series RVUs do not support TNS or accelerated code execution in the OSS environment.

Unless otherwise indicated in the text, all text that applies to systems running H06.14 and later H-series RVUs also applies to systems running J06.03 and later J-series RVUs.

This manual contains information about some of the following G-series development tools. For servers running H-series RVUs, these tools are supported only in H06.05 and subsequent H-series RVUs:

- TNS/R native C compiler
- TNS/R native C++ compiler
- TNS/R native C++ runtime library version 2
- SQL/MP for TNS/R native C
- SQL/MP Compilation Agent for TNS/R programs
- NMCOBOL compiler and **nmcobol** frontend
- **ld**
- **nld**
- **noft**
- TNS/R native pTAL

If your server is running the H06.03 or H06.04 RVU, continue to use the HP Enterprise Toolkit.NonStop Edition or servers running G-series RVUs for development tasks that require these tools. If your server is running J06.03 or later J-series RVUs, these tools are supported.

# Changed Functions

The following reference pages were changed to correct errors:

- **gettimeofday( )**
- **put_awaitio( )**
- **PUT_FILE_OPEN_( )**

The following reference pages were changed to support the increase in message queue limits:

- **msgctl( )**
- **msgget( )**
- **msgrcv( )**
- **msgsnd( )**

# About This Manual

The HP NonStop Open System Services (OSS) application program interface (API) provides an open interface for programs to be run with the underlying HP NonStop operating system.

The *Open System Services System Calls Reference Manual* contains reference pages for OSS system functions, files, and miscellaneous topics.

The description of the OSS API is divided into system functions (documented in this manual) and library functions (documented in the *Open System Services Library Calls Reference Manual*). Functions appear in one manual or the other, based on the logical section number assigned to the reference page for the function. For an explanation of the logical section numbers, see **Reference Section Numbers** later in this section.

This division does not imply any restrictions on the use of the functions described in either manual. The division exists for many reasons, including:

- Consistency with the documentation of the Guardian API. The Guardian API for process management and low-level file-system access is documented separately from other portions of the API available to users of a C run-time library.
- Consistency with the separation of functions used in some UNIX systems. In those systems, an important distinction exists between the API for code that is to run in kernel space and the API for code that is to run in user space. This distinction is meaningless for users of the OSS API. The NonStop operating system does distinguish among code that runs in user code space, code that runs in system code space, and code that runs in library code space, but the distinction does not separate the functions of the OSS API.

Unless otherwise indicated in the text, discussions of native mode behavior, processes, and so forth apply to both the TNS/R code that runs on systems running G-series RVUs and to the TNS/E code that runs on systems running J-series RVUs or H-series RVUs. Discussions of TNS or accelerated code behavior in the OSS environment apply only to systems running G-series RVUs; systems running J-series RVUs or H-series RVUs do not support TNS or accelerated code execution in the OSS environment.

Unless otherwise indicated in the text, all text that applies to systems running H06.14 and later H-series RVUs also applies to systems running J06.03 and later J-series RVUs.

## Audience

This manual is for system and application programmers who want to use the OSS API provided with the NonStop operating system. The manual assumes that the reader is a programmer and is familiar with the C programming language.

## Purpose

This manual provides a complete reference to all OSS system functions and their related files and miscellaneous topics.

## Document Usage

This manual contains a portion of the online reference (man) pages. These reference pages are divided among 12 sections, as follows:

- Sections 1 through 10 contain reference pages for OSS system functions. These reference pages reside in the **cat2** or **cat3** directory and are sorted alphabetically.
- Section 11 contains reference pages for some OSS header and special files. These reference pages reside in the **cat4** and **cat7** directories and are sorted alphabetically.
- Section 12 contains reference pages for some miscellaneous OSS topics. These reference pages reside in the **cat5** directory and are sorted alphabetically.

## Reference Page Format

The reference pages for functions, files, and miscellaneous topics in this manual use the following format. If a heading has no contents for a particular reference page, it is omitted.

**NAME** Function, file, or miscellaneous topic name and purpose.

**LIBRARY** Library containing the function. The library is identified in terms of the run-time environment in which the compiled application must run. For example, an H-series native Guardian process must use the specified library when the **c89 -Wtarget=TNS/E** and **-Wsystype=guardian** flags are specified or used by default.

**SYNOPSIS** Appropriate syntax, including header files to be included and all parameter types. Some header files are required for POSIX.1-compliant applications but are optional for applications conforming to other standards. These header files are noted as "optional except for POSIX.1."

**PARAMETERS** Descriptions of the parameters listed under the **SYNOPSIS** heading.

**DESCRIPTION** For function topics, how the function works, including the conditions or permissions required to use it successfully, the set of values for all parameters, and the effect of the function on the state of processes or files. For file topic reference pages, a description of file contents. For miscellaneous topics, a general description.

**EXAMPLES** Compilable C language program excerpts using the function call described in the reference page.

**NOTES** Any supplementary information that is peripheral to the actual operation of the function, file, or miscellaneous topic described.

**CAUTIONS** Information on possible system damage or data corruption as a result of using the function, file, or miscellaneous topic in a specific way.

**RETURN VALUES** Indication of successful or unsuccessful completion when the function is invoked.

**ERRORS** Error conditions under which the function might fail, and the **errno** value associated with each condition.

**FILES** Files related to the function, file, or miscellaneous topic, except for any header files listed under the **SYNOPSIS** heading.

**RELATED INFORMATION** Cross-references to related functions, files, commands, and miscellaneous topics described in other OSS reference pages. This heading does not contain titles of standards, HP manuals, or commercial texts.

**STANDARDS CONFORMANCE** Summary of features that are fully described under previous headings and are flagged as implementation-defined or HP extensions to the cited standard.

The POSIX standards leave some features to the implementing vendor to define. These features are flagged as implementation-defined. Features that HP has included that are not in the cited standard are flagged as HP extensions to the appropriate cited standard.

## Related Documents

For information about OSS library functions, commands and utilities, and guidelines for general usage, see these manuals:

- *C/C++ Programmer's Guide*
- *Common Run-Time Environment (CRE) Programmer's Guide*

- *eld Manual* (TNS/E systems only)
- *enoft Manual* (TNS/E systems only)
- *H-Series Application Migration Guide*
- *Inspect Manual*
- *ld Manual*
- *Native Inspect Manual* (TNS/E systems only)
- *rld Manual*
- *nld Manual*
- *noft Manual*
- *Open System Services Library Calls Reference Manual*
- *Open System Services Porting Guide*
- *Open System Services Programmer's Guide*
- *Open System Services Shell and Utilities Reference Manual*
- *Open System Services User's Guide*
- *Software Internationalization Guide*
- *TCP/IP and TCP/IPv6 Programming Manual*
- *TNS/R Native Application Migration Guide*

If you are working in or with the Guardian environment, see the *Guardian Procedure Calls Reference Manual* and its related manuals.

## Reference Section Numbers

The online documentation for Open System Services is divided into logical sections. Each logical section has a reference section number.

Some topics in the reference pages have more than one reference page file; a reference section number identifies a specific file. For example, **chown** has a reference page for the **chown( )** function in section 2 and a reference page for the **chown** command in section 1. These topics are identified as **chown(2)** and **chown(1)**, respectively.

The reference section number can be used in the **man** command to select the correct reference page. For more information about the *section* parameter, see the **man** command reference page either online or in the *Open System Services Shell and Utilities Reference Manual*.

Reference section numbers are included under the **RELATED INFORMATION** heading and in the heading at the top of every reference page. The following table shows the correspondence between reference section numbers and OSS manuals.

| Section | Content | Manual |
| --- | --- | --- |
| (1) | User commands | *OSS Shell and Utilities Reference Manual* |
| (2) | System functions | *OSS System Calls Reference Manual* |
| (3) | Library functions | *OSS Library Calls Reference Manual* |
| (4) | File formats and data structures | *OSS System Calls Reference Manual*<br>*OSS Library Calls Reference Manual*<br>*OSS Shell and Utilities Reference Manual* |
| (5) | Miscellaneous topics and environment variables | *OSS System Calls Reference Manual*<br>*OSS Library Calls Reference Manual*<br>*OSS Shell and Utilities Reference Manual* |
| (6) | Games | Not supplied by HP |
| (7) | Special files | *OSS System Calls Reference Manual* |
| (8) | Administrator commands | *OSS Shell and Utilities Reference Manual* |

# Typographic and Keying Conventions

This manual uses the following typographic conventions:

**Bold** — **Bold** words or characters represent system elements that you must use literally, such as commands, flags, and pathnames.

*Italic* — *Italic* words or characters represent variable values that you must supply.

`Constant width` — Examples and information that the system displays appear in the `constant width` typeface.

[ ] — Brackets enclose optional items in format and syntax descriptions.

| — A vertical bar separates items in a list of choices.

... — A horizontal ellipsis indicates that you can repeat the preceding item one or more times. A vertical ellipsis indicates that you can repeat the preceding line one or more times.

In text margins, a vertical bar indicates a line changed since the last revision of the reference page.

# Section 1. System Functions (a - d)

This section contains reference pages for Open System Services (OSS) system function calls with names that begin with **a** through **d**. These reference pages reside in the **cat2** directory and are sorted alphabetically by U.S. English conventions in this section.

## NAME

**accept -** Accepts a new connection on a socket

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int accept(
        int socket,
        struct sockaddr *address,
        socklen_t *address_len);
```

## PARAMETERS

*socket*
: Specifies the file descriptor for a socket that was created with the **socket( )** function, has been bound to an address with the **bind( )** function, and has issued a successful call to the **listen( )** function.

*address*
: Specifies either a null pointer or a pointer to the **sockaddr** structure where the address of the peer socket that requested the connection should be returned. The length and format of the address depend on the address family of the socket.

For **AF_INET** sockets, a pointer to the address structure **sockaddr_in** must be cast as a **struct sockaddr**. For **AF_INET6** sockets, a pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**. For **AF_UNIX** sockets, a pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*address_len*
: Points to a **socklen_t** data item, which, on input, specifies the length of the **sockaddr** structure pointed to by the *address* parameter, and, on output, specifies the length of the address returned.

## DESCRIPTION

The **accept( )** function extracts the first connection on the queue of pending connections, creates a new socket with the same socket type, protocol, and address family as the specified socket, and allocates a new file descriptor for that socket.

In systems running **AF_UNIX** Release 2 software, the new socket will use the same mode (compatibility or portability) as the specified *socket*.

For more information about **AF_UNIX** Release 2, compatibility mode, and portability mode, see the *Open System Services Programmer's Guide*.

When the **accept( )** function is called using a value for the *address* parameter that is null, successful completion of the call returns a socket file descriptor without modifying the value pointed to by the *address_len* parameter. When the **accept( )** function is called using a value for the *address* parameter that is not null, a successful call places the address of the peer socket in the **sockaddr** structure pointed to by the *address* parameter, and places the length of the peer socket's address in the location pointed to by the *address_len* parameter.

If the length of the socket address is greater than the length of the supplied **sockaddr** structure, the address is truncated when stored.

If the queue of pending connections is empty of connection requests and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **accept( )** function blocks until a connection is present. If the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the queue of pending connections is empty, the **accept( )** function call fails and sets **errno** to [EWOULDBLOCK].

**NOTES**

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When a connection is available, a call to the **select( )** function indicates that the file descriptor for the original socket is ready for reading.

The accepted socket cannot itself accept more connections. The original socket remains open and can accept more connections.

To use the **accept( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_acceptx(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **accept( )** function returns the file descriptor of the accepted socket. If the **accept( )** function call fails, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **accept( )** function sets **errno** to the corresponding value:

[EADDRINUSE]
The address is already in use. This error is returned in the OSS environment only.

[EBADF] The *socket* parameter is not a valid file descriptor.

This error is also returned if the **accept( )** function is thread-aware and the socket becomes invalid (is closed by another thread).

[ECONNABORTED]
The connection was aborted.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINTR] The function call was interrupted by a signal that was caught before a valid connection arrived.

This error is also returned if the **accept( )** function is thread-aware and a signal received from the **pthread_kill( )** function is not blocked, ignored, or handled.

[EINVAL] The socket is not accepting connections.

[EMFILE] No more file descriptors are available for this process.

[ENFILE] One of these conditions exists:

- The maximum number of file descriptors of this file type (socket, pipe, etc.) for this processor are already open.
- The limit for open file descriptors of this file type has not been exceeded, but the maximum number of all file descriptors for this processor are already open.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time might succeed.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTSOCK] The *socket* parameter does not specify a socket.

[EOPNOTSUPP]
The socket type of the specified socket does not support accepting connections.

[EWOULDBLOCK]
The socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set) and no connections are present to be accepted.

**RELATED INFORMATION**

Functions: **bind(2)**, **connect(2)**, **fcntl(2)**, **listen(2)**, **socket(2)**, **spt_acceptx(2)**.

**STANDARDS CONFORMANCE**

The XPG4 specification allows certain behaviors to be implementer-defined. The following are choices of the HP implementation:

- The HP implementation does not return the **errno** values [EAGAIN], [ENOSR], or [EPROTO].

The following are HP extensions to the XPG4 specification:

- The **errno** value [ECONNRESET] can be returned when the transport-provider process is unavailable.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**access -** Determines the accessibility of a file

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <unistd.h>**

**int access(**
**const char ****path***,**
**int** *access_mode***);**

## PARAMETERS

*path* Points to the file pathname. When the *path* parameter refers to a symbolic link, the **access( )** function returns information about the file pointed to by the symbolic link.

Permission to access all components of the *path* parameter is determined by using a real user ID instead of an effective user ID, and by using a real group ID instead of an effective group ID.

*access_mode* Specifies the type of access. The bit pattern contained in the *access_mode* parameter is constructed by a logical OR of values from the following list:

**F_OK** Checks to see whether the file exists.

**R_OK** Checks read permission.

**W_OK** Checks write permission.

**X_OK** Checks execute (search) permission.

## DESCRIPTION

The **access( )** function checks the accessibility of a file specified by a pathname.

Only access bits are checked. A directory can be indicated as writable by **access( )**, but an attempt to open it for writing could fail (although files can be created there).

A process with appropriate privilege can override the file permissions of a file. For files in unrestricted filesets, a process with the super ID has the appropriate privilege. However, for files in restricted-access filesets, such special access privileges are restricted even for the super ID. For information about restricted-access filesets, see the *Open System Services Management and Operations Guide*.

### Access Control Lists (ACLs)

Read, write, and execute/search permissions are checked against the ACL for the file.

To determine the permission granted to the real user ID (RUID) and real group ID (RGID) of the accessing process, the **access( )** function checks these things in the ACL, in order:

1. If the RUID of the process is the same as the owner of the file, grant the permissions specified in the **user::** entry. Otherwise, continue to the next check.

2. If the RUID matches the UID specified in one of the additional **user:uid:** entries, grant the permissions specified in that entry, bitwise-ANDed with the permissions specified in the **class** entry. Otherwise, continue to the next check.

3. If the RGID or a supplementary GID of the process matches the owning GID of the file or one of the GIDs specified in any additional **group:gid:** entries, grant the permissions

specified in the class entry bitwise-ANDed with the result of bitwise-ORing together all of the permissions in all matching group entries. Otherwise, continue to the next check.

4. Otherwise, grant the permissions specified in the **other** entry.

### Files in the Guardian File System

If the specified pathname resolves to the **/G** directory itself, the calling process has read and execute access but not write access. The permissions are "r-xr-xr-x".

If the specified pathname resolves to a Guardian process name, the calling process has execute access but not read or write access. The permissions are "--x--x--x".

If the specified pathname resolves to a Guardian disk volume or subvolume, then the calling process has read, write, and execute access. The permissions are "rwxrwxrwx".

If the specified pathname resolves to a regular Guardian disk file, then Guardian standard security and Safeguard file-level protection govern access. Refer to the **stat(2)** reference page for more information.

### Use From the Guardian Environment

The **access( )** function is one of a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **access( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **access( )** function sets **errno** to the corresponding value:

[EACCES] The permission bits of the file mode do not permit the requested access, or search permission is denied on a component of the pathname prefix. The owner of a file has permissions checked with respect to the "owner" read, write, and execute mode bits; members of the file's group other than the owner have permissions checked with respect to the "group" mode bits; and all others have permissions checked with respect to the "other" mode bits.

[EFAULT] The *path* parameter points outside the process's allocated address space.

[EFSBAD] The program attempted an operation involving a fileset with a corrupted fileset catalog.

[EINTR] A signal was caught during execution of the function call.

[EINVAL] The *access_mode* parameter contains an invalid bit pattern.

[EIO] An I/O error occurred during a read from or a write to the fileset.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of the following conditions exists:

- The specified pathname does not exist.
- The specified pathname is an empty string.
- The specified pathname cannot be mapped to a valid Guardian filename.
- The *path* parameter specifies a file on a remote HP NonStop node but communication with the remote node has been lost.

[ENOROOT] One of the following conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node and communication with the remote name server has been lost.

[ENOTDIR] A component of the pathname prefix is not a directory.

[ENOTSUP] The *path* parameter specifies a Guardian file on an SMF logical volume and one of the following conditions exists:

- The local system is running an RVU prior to J06.15 or H06.26.
- The *path* parameter specifies a file in /E and the remote system is running an RVU prior to J06.15 or H06.26.

[ENXIO] The fileset containing the current working directory or the root fileset is not mounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] Write access is requested for a file on a read-only fileset.

[ETXTBSY] Write access is requested for a pure procedure (shared text) file that is being executed.

**RELATED INFORMATION**

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **stat(2)**.

Miscellaneous topics: **acl(5)**.

**STANDARDS CONFORMANCE**

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- A process with appropriate privilege can override the file permissions of a file. For files in unrestricted filesets, a process with the super ID has the appropriate privilege. However, for files in restricted-access filesets, such special access privileges are restricted even for the super ID.

- The error [EINVAL] can be detected.

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EFSBAD], [EINTR], [EIO], [ENOROOT], [ENOTSUP], [ENXIO], and [EOSSNOTRUNNING] can be returned.

## NAME

**acl -** Sets access control list (ACL) information for a file

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/acl.h>
int acl(
    char *pathp, int cmd, int nentries,
    struct acl *aclbufp);
```

## PARAMETERS

*pathp* — Points to the pathname of the file.

*nentries* — Specifies the number of ACL entries pointed to by the *aclbufp* parameter.

*aclbufp* — Points to the first element of a structure of type **acl**. The **acl** structure is defined in the **acl.h** header file as:

```
struct acl {
        int            a_type;  /* entry type */
        uid_t          a_id;    /* user or group ID */
        unsigned short a_perm;  /* entry permissions */
};
```

The values of the **a_type** field are:

**USER_OBJ** Permissions for the owner of the object

**USER** Permissions for additional specified users

**GROUP_OBJ** Permissions for members of the owning group of the object

**GROUP** Permissions for members of additional specified groups

**CLASS_OBJ** Maximum permissions granted to the file group class

**OTHER_OBJ** Permissions for other users

**DEF_USER_OBJ**
Default permissions for the object owner

**DEF_USER** Default permissions for additional specified users

**DEF_GROUP_OBJ**
Default permissions for members of the owning group of the object

**DEF_GROUP** Default permissions for members of additional specified groups

**DEF_CLASS_OBJ**
Default maximum permissions for the owning group, additional specified users, and additional specified groups.

**DEF_OTHER_OBJ**
Default permissions for other users

*cmd* Specifies the action to be taken by the **acl( )** function. The *cmd* parameter can be one of these values:

**ACL_SET** The **acl( )** function stores the entries specified by the *nentries* and *aclbufp* parameters in the ACL for the file. The new ACL replaces any existing ACL for the file. This value for *cmd* can only be executed by a process that has an effective user ID equal to the owner of the file or the super ID, or is a member of the Safeguard SECURITY-OSS-ADMINISTRATOR group. All directories in the pathname must be searchable.

**ACL_GET** The buffer *aclbufp* is filled with the ACL entries for the file. Discretionary read access to the file is not required, but all directories in the pathname must be searchable.

**ACL_CNT** The number of entries in the ACL for the file is returned. Discretionary read access to the file is not required, but all directories in the pathname must be searchable.

## DESCRIPTION

The **acl( )** function manipulates ACLs on file system objects in filesets that support OSS ACLs. A process on a system that does not support ACLs can use the **chmod( )** function to remotely modify the permissions in the base ACL entries of a file (see the **chmod(2)** reference page). ACLs are supported for OSS files only. For a detailed description of ACLs, see the **acl(5)** reference page.

A call to **acl( )** specified with the **ACL_SET** command succeeds only if all of these conditions are true:

- The ACL contains exactly one entry each of type **USER_OBJ**, **GROUP_OBJ**, **CLASS_OBJ**, and **OTHER_OBJ**.
- If *pathp* points to a directory, the ACL contains at most one entry each of type **DEF_USER_OBJ**, **DEF_GROUP_OBJ**, **DEF_CLASS_OBJ**, and **DEF_OTHER_OBJ**.
- Entries of type **USER**, **GROUP**, **DEF_USER**, or **DEF_GROUP** do not contain duplicate entries. A duplicate entry is one of the same type containing the same numeric ID.
- If the ACL contains no entries of type **USER** and no entries of type **GROUP**, the entries of type **GROUP_OBJ** and **CLASS_OBJ** have the same permissions.
- If the ACL contains no entries of type **DEF_USER** and no entries of type **DEF_GROUP**, and an entry of type **DEF_GROUP_OBJ** is specified, an entry of type **DEF_CLASS_OBJ** is also specified and the two entries have the same permissions.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted by the same file permissions and owner privileges as any other user ID: It has no special privileges unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

Processes that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit applications or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **acl( )** function returns one of the following values, depending on the value of the *cmd* parameter:

- For successful **ACL_CNT** or **ACL_GET** commands, the **acl( )** function returns the number of ACL entries.
- For successful **ACL_SET** commands, the **acl( )** function returns a 0 (zero).

If the **acl( )** function fails, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **acl( )** function sets **errno** to the corresponding value:

[EACCES] The caller does not have access to a component of the pathname.

[EINVAL] One of these conditions occurred:

- The value of the *cmd* parameter is not **ACL_GET**, **ACL_SET**, or **ACL_CNT**.
- The value of the *cmd* parameter is **ACL_SET**, and the value of the *nentries* parameter is less than the number of mandatory ACL entries.
- The value of the *cmd* parameter is **ACL_SET**, and the ACL specified in the *aclbufp* parameter is not valid.

[EIO] A disk I/O error occurred while attempting to store or retrieve the ACL

[EPERM] One of the following conditions exist:

- The value of the *cmd* parameter is **ACL_SET**, and the effective user ID of the caller does not match the owner of the file or the super ID, and the effective user ID or one of its group affiliations does not qualify the caller for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[ENOENT] A component of the path does not exist.

[ENOSPC] One of these conditions occurred:

- The value of the *cmd* parameter is **ACL_GET**, and the value of the *nentries* parameter is less than the number of entries in the ACL for the file.
- The value of the *cmd* parameter is **ACL_SET**, and there is not enough disk space to store the ACL.
- The value of the *cmd* parameter is **ACL_SET**, and *nentries* is greater than the number of **NACLENTRIES** specfied in the **acl.h** header file.

[ENOSYS] This system does not support OSS ACLs.

[ENOTDIR] Either of these conditions is true:

- A component of the path specified by *pathp* is not a directory.
- The value of the *cmd* parameter is **ACL_SET**, and an attempt was made to set a default ACL on a file type other than a directory.

[ENOTSUP] The file specified by *pathp* either resides in a fileset that does not support OSS ACLs, or is a file type that does not support OSS ACLs (such as Guardian files accessed from the **/G** directory).

[EROFS] The value of the *cmd* parameter is **ACL_SET**, and the file specified by *pathp* resides in a fileset that is mounted as read-only.

[EFAULT] The *aclbufp* parameter points to an invalid address.

## RELATED INFORMATION

Commands: **getacl(1)**, **setacl(1)**.

Functions: **aclsort(3)**, **creat(2)**, **creat64(2)**, **open(2)**, **open64(2)**.

Miscellaneous: **acl(5)**.

## STANDARDS CONFORMANCE

This function is an HP extension to the XPG4 Version 2 specification.

## NAME

**bind** - Binds a name to a socket

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <sys/socket.h>**

**int bind(**
    **int** *socket***,**
    **const struct sockaddr** ***address*,**
    **socklen_t** *address_len***);**

## PARAMETERS

*socket* Specifies the file descriptor of the socket to be bound.

*address* Points to a **sockaddr** structure that contains the address to be bound to the socket. The length and format of the address depend on the address family of the socket.

For **AF_INET** sockets, a pointer to the address structure **sockaddr_in** must be cast as a **struct sockaddr**. For **AF_INET6** sockets, a pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**. For **AF_UNIX** sockets, a pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*address_len* Specifies the length of the **sockaddr** structure pointed to by the *address* parameter.

## DESCRIPTION

The **bind( )** function assigns a name, which consists of an address stored in a **sockaddr** structure, to an unnamed socket. Sockets created with the **socket( )** function are initially unnamed; they are identified only by their address family.

An application program can retrieve the assigned socket name with the **getsockname( )** function.

### Access Control Lists (ACLs)

If the parent directory has an ACL that contains default ACL entries, **bind( )** creates an ACL for the socket that inherits the default ACL entries of the parent directory as actual ACL entries for the socket. For more information about ACL inheritance, see the **acl(5)** reference page.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted by the same file permissions and owner privileges as any other user ID: It has no special privileges unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

In a restricted-access fileset, applications that have the PRIVSOARFOPEN privilege and are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in the fileset. However, Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **bind( )** function returns the value 0 (zero). Otherwise, the **bind( )** function returns -1 and sets **errno** to indicate the error.

## ERRORS

If any of the following conditions occurs, the **bind( )** function sets **errno** to the corresponding value:

[EACCES] One of the following conditions occurred:

- The specified address is protected and the current user does not have permission to bind to it.
- The socket is an **AF_UNIX** socket and either a component of the path prefix denies search permission, or the requested name requires writing in a directory with a mode that denies write permission.

[EADDRINUSE]
The specified address is already in use.

[EADDRNOTAVAIL]
The specified address is not available on this HP NonStop node.

[EAFNOSUPPORT]
The specified address is not a valid address for the address family of the specified socket.

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EDESTADDRREQ]
The *address* parameter for an **AF_UNIX** socket is a null pointer.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EINVAL] One of the following conditions exists:

- The socket is already bound to an address.
- The socket has been shut down.
- The size specified for the *address_len* parameter is not valid for an address in the address family that is used by this connection.

[EIO] An input or output error occurred for an **AF_UNIX** socket.

[ELOOP] The socket is in the **AF_UNIX** domain and too many symbolic links were encountered when translating the pathname specified in the **sockaddr** structure.

[ENAMETOOLONG]
The socket is in the **AF_UNIX** domain and one of the following conditions exists:

- The pathname in the **sockaddr** structure exceeds **PATH_MAX** characters.
- A component of the pathname in the **sockaddr** structure exceeds **NAME_MAX** characters.
- The intermediate result of pathname resolution when a symbolic link is part of the pathname in the **sockaddr** structure exceeds **PATH_MAX** characters.

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] The socket is in the **AF_UNIX** domain and one of the following conditions exists:

- A component of the pathname specified in the **sockaddr** structure does not name an existing file.
- The **sockaddr** structure specifies an empty string as a pathname.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time might succeed.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOTDIR] The socket is in the **AF_UNIX** domain and a component of the pathname specified in the **sockaddr** structure is not a directory.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The socket type of the specified socket does not support binding to an address.

[EPERM] The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The socket is in the **AF_UNIX** domain and the specified name would reside on a read-only fileset.

**RELATED INFORMATION**

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **connect(2)**, **getsockname(2)**, **listen(2)**, **socket(2)**.

Miscellaneous topics: **acl(5)**.

## STANDARDS CONFORMANCE

The XPG4 specification allows certain behaviors to be implementer-defined. The following are choices of the HP implementation:

- The HP implementation does not return the **errno** values [EISCONN], [EISDIR], [ENOSR], or [EPROTO].

The following are HP extensions to the XPG4 specification:

- The **errno** value [ECONNRESET] can be returned when the transport provider process is unavailable.

## NAME

**chdir -** Changes the current working directory

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <unistd.h>**

**int chdir(**
**const char ***_path_**);**

## PARAMETERS

*path* Points to the pathname of the directory.

## DESCRIPTION

The **chdir( )** function changes the current working directory to the directory indicated by the *path* parameter. If the *path* parameter refers to a symbolic link, the **chdir( )** function sets the current directory to the directory pointed to by the symbolic link.

The current working directory is the starting point for searches for pathnames that do not begin with a **/** (slash). For a directory to become the current working directory, the calling process must have search (execute) access to the directory.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted by the same file permissions and owner privileges as any other user ID: It has no special privileges unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

Processes that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use on Guardian Objects

Guardian process names are directories; however, they cannot be opened using **chdir( )**. Attempts to do so fail and set **errno** to the value [EPERM].

A call to the **chdir( )** function with a *path* parameter that points to a subprocess in the Guardian file system fails when the process is not of subtype 30. Such a call sets **errno** to the value [ENOENT].

A call to the **chdir( )** function with a *path* parameter that points to an empty Guardian disk subvolume (for example, **/G/vol/subvol**) succeeds.

A call to the **chdir( )** function with a *path* parameter that points to a Guardian subvolume with a reserved name (for example, **/G/vol1/zyq00001**) fails. Such a call sets **errno** to the value [EPERM].

### Use From the Guardian Environment

The **chdir( )** function is one of a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

A process running with an effective user ID or group affiliation that qualifies for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group has read and search permissions for any OSS directory.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **chdir( )** function returns the value 0 (zero). Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **chdir( )** function sets **errno** to the corresponding value:

[EACCES] The requested current working directory is not accessible because search permission is denied for a component of the pathname.

[EFAULT] The *path* parameter is an invalid address.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EIO] A physical input or output error occurred.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG]
One of these is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of these conditions exists:

- The named directory does not exist.
- The specified pathname is an empty string.

- The specified pathname cannot be mapped to a valid Guardian filename.
- The specified pathname points to the name of a Guardian process that is not of subtype 30.
- The *path* parameter names a symbolic link, but the directory to which it refers does not exist.
- The *path* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOROOT] One of these conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node, and communication with the remote name server has been lost.

[ENOTDIR] A component of the pathname is not a directory.

[ENXIO] The fileset containing the client's current working directory or root directory is not mounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] One of the following conditions exist:

- The program attempted an operation on a Guardian process or attempted to access a Guardian ZYQ subvolume.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

**RELATED INFORMATION**

Commands: **cd(1)**.

Functions: **chroot(2)**.

Miscellaneous topics: **acl(5)**.

**STANDARDS CONFORMANCE**

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EFSBAD], [EIO], [ENOROOT], [ENXIO], [EOSSNOTRUNNING], and [EPERM] can be returned.

## NAME

**chmod -** Changes file-access permissions

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <sys/stat.h>

int chmod(
        const char *path,
        mode_t mode);
```

## PARAMETERS

*path* Specifies the full pathname of the file. If the *path* parameter refers to a symbolic link, the **chmod( )** function changes access permissions on the file specified by the symbolic link.

*mode* Specifies the bit pattern that determines the access permissions.

## DESCRIPTION

The **chmod( )** function sets the access permissions of the file specified by the *path* parameter according to the bit pattern specified by the *mode* parameter.

To change the file access permissions of a file or directory, the effective user ID of the process must match the super ID or the owner of the file, or its effective user ID or one of its group affiliations must qualify it for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group.

If **chmod( )** is invoked by a process whose effective user ID does not equal the super ID or file owner, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000, respectively) are cleared.

If **chmod( )** is invoked to set either or both of the set-user-ID and set-group-ID bits of the file mode (04000 and 02000 respectively), then any file privileges the file might have had are cleared.

See also "Accessing Files in Restricted-Access Filesets."

If the **S_ISVTX** bit is on for a directory, only processes with an effective user ID equal to the user ID of the file's owner or the directory's owner, or a process with appropriate privileges, can remove files from the directory.

A call to the **chmod( )** function has no effect on the file descriptor for a file that is open at the time of the call. However, new openers of the file will be authorized by using the new access permissions that were specified in the call.

The *mode* parameter is constructed by logically ORing one or more of these symbols, which are defined in the **sys/stat.h** header file:

**S_ISUID** Sets the process's effective user ID to the user ID of the file's owner on execution.

**S_ISGID** Sets the process's effective group ID to the group ID of the file's group on execution.

**S_ISVTX** For a directory, permits modification to the directory only if the effective user ID of the process matches that of the file being accessed.

**S_IRWXU** Permits the file's owner to read, write, and execute the file (or to search the directory).

**S_IRUSR** Permits the file's owner to read the file.

**S_IWUSR** Permits the file's owner to write to the file.

**S_IXUSR** Permits the file's owner to execute the file (or to search the directory).

**S_IRWXG** Permits the file's group to read, write, and execute the file (or to search the directory).

**S_IRGRP** Permits the file's group to read the file.

**S_IWGRP** Permits the file's group to write to the file.

**S_IXGRP** Permits the file's group to execute the file (or to search the directory).

**S_IRWXO** Permits others to read, write, and execute the file (or to search the directory).

**S_IROTH** Permits others to read the file.

**S_IWOTH** Permits others to write to the file.

**S_IXOTH** Permits others to execute the file (or to search the directory).

**S_TRUST** Establishes that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers when the memory segment containing the buffers is not shared. This flag applies only to loadfiles for a TNS/E native process and can be set only by a user with appropriate privileges (the super ID).

**S_TRUSTSHARED**
Establishes that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers regardless of whether the memory segment containing the buffers is shared. This flag applies only to loadfiles for a TNS/E native process and can be set only by a user with appropriate privileges (the super ID).

If the file specified by the *path* parameter resides on an HP NonStop node where the calling process is not logged in, the **S_ISUID** bit of the file is cleared by the call, but the **S_ISGID** bit of the file is not cleared by the call.

The **S_ISGID** bit of the file is cleared if all of these conditions are true:

- The named file is a regular file.
- The process does not have appropriate privileges.
- The file's group ID does not match the effective group ID of the process or one of the IDs of the process's group list.

Upon successful completion, the **chmod( )** function marks the **st_ctime** field of the file for update.

### Access Control Lists (ACLs)

When you execute the **chmod( )** function, you can change the effective permissions granted by optional entries in the ACL for a file. In particular, using the **chmod( )** function to remove read, write, and execute permissions from a file owner, owning group, and all others works as expected, because the **chmod( )** function affects the **class** entry in the ACL, limiting any access that can be granted to additional users or groups through optional ACL entries. To verify the effect, use **getacl** command on the file after the **chmod( )** function completes and note that all optional (nondefault) ACL entries with nonzero permissions also have the comment **# effective:---**.

To set the permission bits of ACL entries, use the **acl( )** function instead of the **chmod( )** function.

ACLs are not supported for symbolic links.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted. In a restricted-access fileset:

- The super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is not permitted to invoke this function on files that it does not own unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

- Processes that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, if the executable file for the process does not have the PRIVSOARFOPEN file privilege, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000 respectively) of the file accessed by this function are cleared.

- Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use on Guardian Objects

Attempting to set the access permissions on a Guardian file (that is, a file in the **/G** file system) fails with **errno** set to [EINVAL].

### Use From the Guardian Environment

The **chmod( )** function is one of a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.

- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.

- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **chmod( )** function returns the value 0 (zero). Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occur, the **chmod( )** function sets **errno** to the corresponding value:

[EACCES] Search permission is denied for a component of the *path* parameter.

[EFAULT] The *path* parameter points to a location outside of the allocated address space of the process.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EINVAL] One of these conditions exists:

- The value of the *mode* parameter is invalid.
- An attempt was made to set access permissions on a Guardian file (that is, a file in the **/G** file system).

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ELOOP] Too many symbolic links were encountered in translating the *path* parameter.

[ENAMETOOLONG]
One of these is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of these conditions exists:

- The named file does not exist, or the specified name is an empty string.
- The *path* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOROOT] One of these conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node, and communication with the remote name server has been lost.

[ENOTDIR] A component, other than the last part, of the *path* parameter is not a directory.

[ENXIO] The fileset containing the client's current working directory or root directory is not mounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] One of the following conditions exist:

- The effective user ID does not match the user ID of the owner of the file, or the owner does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The named file resides on a read-only fileset.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Commands: **chmod(1)**, **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chown(2)**, **fchmod(2)**, **fchown(2)**, **fcntl(2)**, **getgroups(2)**, **lchmod(2)**, **lchown(2)**, **mknod(2)**, **open(2)**, **open64(2)**, **read(2)**, **setfilepriv(2)**, **write(2)**.

Miscellaneous topics: **acl(5)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- To change file-access permissions, either the process must have the same effective user ID as the owner of the file or the process must have an effective user ID of the super ID.
- A call to the **chmod( )** function has no effect on the file descriptor for a file that is open at the time of the call. However, new openers of the file are authenticated by using the new access permissions that were specified in the call.

HP extensions to the XPG4 Version 2 specification are:

- To change the file access permissions of a file or directory, the effective user ID of the process must match the super ID or the owner of the file, or the effective user ID or one of the group affiliations for the process must qualify the process for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group.
- The **errno** values [EFAULT], [EFSBAD], [EINVAL], [ENOROOT], [ENXIO], and [EOSSNOTRUNNING] can be returned.

## NAME

**chown -** Changes the owner and group IDs of a file

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>

int chown(
        const char *path,
        uid_t owner,
        gid_t group);
```

## PARAMETERS

*path* Specifies the name of the file whose owner ID, group ID, or both are to be changed. If the final component of the *path* parameter names a symbolic link, the link is traversed, and pathname resolution continues.

When the *path* parameter refers to a symbolic link, the **chown( )** function changes the ownership of the file pointed to by the symbolic link.

*owner* Specifies a numeric value representing the owner ID.

*group* Specifies a numeric value representing the group ID.

## DESCRIPTION

The **chown( )** function changes the owner and group of a file.

Only a process that has an effective user ID equal to the super ID or to the file owner, or that has an effective user ID or group affiliation qualifying for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group can use the **chown( )** function to change the group of a file. However, processes that have an effective user ID equal to the file owner can only change the group of a file to a group to which they belong (their effective group or one of their supplementary groups).

If the **chown( )** function is invoked by a process whose effective user ID does not equal the super ID, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000, respectively) are cleared.

See also "Accessing Files in Restricted-Access Filesets."

If the **chown( )** function is successfully invoked on a file, the **S_ISGID** and **S_ISUID** bits of the **st_mode** field of the **stat** structure are cleared unless the user has appropriate privileges.

The **_POSIX_CHOWN_RESTRICTED** feature is enforced for any file in the OSS file system. Only processes with appropriate privileges can change owner IDs.

If the *owner* or *group* parameter is specified as -1 cast to the type of **uid_t** or **gid_t**, respectively, the corresponding ID of the file is unchanged. To change only one attribute, specify the other as -1.

Upon successful completion, the **chown( )** function marks the **st_ctime** field of the file for update.

### Access Control Lists (ACLs)

A user can allow or deny specific individuals and groups access to a file by using an ACL on the file. When using the **chown( )** function with ACLs, if the new owner and/or group of a file have optional ACL entries corresponding to **user:***uid***:***perm* or **group:***gid***:***perm* in the ACL for a file, those entries remain in the ACL but no longer have any effect because they are superseded by the **user::***perm* or **group::***perm* entries in the ACL.

ACLs are not supported for symbolic links.

For more information about ACLs, see the **acl(5)** reference page.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted. In a restricted-access fileset:

- The super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is not permitted to invoke this function on files that it does not own unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.
- Processes that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, if the executable file for the process does not have the PRIVSOARFOPEN file privilege, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000 respectively) of the file accessed by this function are cleared.
- Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use on Guardian Objects

The **chown( )** function can be used on Guardian disk files (that is, disk files in the **/G** file system). Attempts to change the ownership of other types of Guardian files fail and set **errno** to [EINVAL].

For Guardian disk files, Guardian security is used, and any user can pass file ownership to any other user. A value other than -1 must be specified for the *owner* parameter (that is, an owner ID must be specified). However, changing the owner ID also changes the group ID to the Guardian group ID of the new owner (that is, bits <16:23> of the new access ID). The **chown( )** function cannot be used to set the group ID for a Guardian file except as a result of changing the owner ID.

The **_POSIX_CHOWN_RESTRICTED** feature is ignored for files in the Guardian file system (that is, for files in **/G**).

### Use From the Guardian Environment

The **chown( )** function is one of a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **chown( )** function returns the value 0 (zero). Otherwise, the value -1 is returned, the owner and group of the file remain unchanged, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occur, the **chown( )** function sets **errno** to the corresponding value:

[EACCES] Search permission is denied on a component of the *path* parameter.

[EFAULT] The *path* parameter is an invalid address.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EINVAL] The *owner* or *group* parameter is out of range.

An attempt was made to change ownership of a Guardian file that is not a disk file.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ELOOP] Too many links were encountered in translating the *path* parameter.

[ENAMETOOLONG]

One of these is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of these is true:

- The *path* parameter does not exist.
- The *path* parameter is an empty string.
- The *path* parameter specifies a file in the Guardian file system (in **/G**) but cannot be mapped to a valid Guardian filename.
- The *path* parameter names a symbolic link, but the file to which it refers does not exist.
- The *path* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOTSUP] The *path* parameter specifies a Guardian file on an SMF logical volume and one of the following conditions exists:

- The local system is running an RVU prior to J06.15 or H06.26.
- The *path* parameter specifies a file in /E and the remote system is running an RVU prior to J06.15 or H06.26.

[ENOROOT] One of these conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node, and communication with the remote name server has been lost.

[ENOTDIR] A component of *path* is not a directory.

[ENXIO] The fileset containing the client's current working directory or root directory is not mounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The named file resides on a read-only fileset.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Commands: **chgrp(1)**, **chown(1)**, **getacl(1)**, **setacl(1)**.

Functions: **acl(2) chmod(2)**, **fchmod(2)**, **fchown(2)**, **lchmod(2)**, **lchown(2)**, **setfilepriv(2)**.

Miscellaneous topics: **acl(5)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- A process can change the value of the owner ID of a file only if the effective user ID of the process gives the process appropriate privileges.
- Upon successful completion, the set-user-ID attribute (the **S_ISUID** bit) and the set-group-ID attribute (the **S_ISGID** bit) of the file are always cleared.
- The error [EINVAL] can be detected.

HP extensions to the XPG4 Version 2 specification are:

- To change the file access permissions of a file or directory, the effective user ID of the process must match the super ID or the owner of the file, or the effective user ID or one of the group affiliations for the process must qualify the process for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group.
- The **errno** values [EFAULT], [EFSBAD], [EIO], [ENOROOT], [ENOTSUP], [ENXIO], and [EOSSNOTRUNNING] can be returned.

## NAME

**chroot -** Changes the effective root directory

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <unistd.h>**

**int chroot(**
  **const char** ***path*);**

## PARAMETERS

*path* Specifies the new effective root directory. If the *path* parameter refers to a symbolic link, the **chroot( )** function sets the effective root directory to the directory pointed to by the symbolic link.

The *path* parameter cannot specify **/E**, and the current working directory of the calling process cannot be a directory in **/E**. If either condition is not met, the call fails and **errno** is set to the value [EINVAL].

## DESCRIPTION

The **chroot( )** function causes the directory named by the *path* parameter to become the effective root directory. The effective root directory is the starting point when searching for a file with an absolute pathname.

The current working directory is not changed by a call to the **chroot( )** function. However, if an absolute pathname is specified in a subsequent call to the **chdir( )** function, that pathname is resolved using the effective root directory.

The calling process must have appropriate privileges in order to change the effective root directory. The calling process must also have search access to the new effective root directory.

The **..** (dot-dot) entry in the effective root directory is interpreted to mean the effective root directory itself. Thus, **..** (dot-dot) cannot be used to access files outside the subtree rooted at the effective root directory.

### Use on Guardian Objects

The *path* parameter can specify **/G** or any volume or subvolume in **/G**.

Guardian process names are directories; however, they cannot be opened using **chroot( )**. Attempts to do so fail and set **errno** to the value [EPERM].

A call to the **chroot( )** function with a *path* parameter that points to a subprocess in the Guardian file system fails when the process is not of subtype 30. Such a call sets **errno** to the value [ENOENT].

A call to the **chroot( )** function with a *path* parameter that points to an empty Guardian disk subvolume (for example, **/G/vol/subvol**) succeeds.

A call to the **chroot( )** function with a *path* parameter that points to a Guardian subvolume with a reserved name (for example, **/G/vol1/zyq00001**) fails. Such a call sets **errno** to the value [EPERM].

### Use From the Guardian Environment

The **chroot( )** function is one of a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file-system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

Use of this function can make an application difficult to port to another system.

If the effective root directory is not **/** (the local node root directory), all files in **/E** become unavailable to the program when the call is completed.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the effective root directory remains unchanged and the **chroot( )** function sets **errno** to the corresponding value:

[EACCES] Search permission is denied for any component of the pathname.

[EFAULT] The *path* parameter points outside the process's allocated address space.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EINVAL] One of the following conditions exists:

- The *path* parameter specifies a Guardian process in **/G**.
- The *path* parameter specifies a file in **/E**.
- The current working directory of the calling process is in **/E**.

[ELOOP] More than **MAXSYMLINKS** symbolic links were encountered while resolving *path*.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of the following conditions exists:

- The named directory does not exist.
- The specified pathname is an empty string.
- The specified pathname cannot be mapped to a valid Guardian filename.

[ENOROOT] The root fileset (fileset 0) is not in the STARTED state.

[ENOTDIR] A component of the specified pathname is not a directory.

[ENXIO] The fileset containing the client's working directory or effective root directory is not mounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] One of the following conditions exists:

- The *path* parameter specifies a subvolume in **/G** with a reserved name (for example, **/G/***volume***/ZYQ00000**).
- The *path* parameter specifies a process name in **/G** (for example, **/G/ztnt**).
- The *path* parameter specifies an invalid subvolume name in **/G**.
- The effective user ID of the process is not the root ID and does not have appropriate privileges to change the root directory.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

## RELATED INFORMATION

Commands: **cd(1)**.

Functions: **chdir(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EFSBAD], [EINVAL], [ENOROOT], [ENXIO], and [EOSSNOTRUNNING] can be returned.

## NAME

**close -** Closes a file descriptor

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <unistd.h>

int close(
        int filedes);
```

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function, or the thread-aware **accept( )**, **dup2( )**, or **fcntl( )** function.

## DESCRIPTION

The **close( )** function closes the file descriptor specified by the *filedes* parameter.

All regions of the file associated with the *filedes* parameter that this process has previously locked with the **fcntl( )** function are unlocked. This occurs even if the process still has the file open by another file descriptor.

When the last file descriptor associated with an open file descriptor is closed:

- The open file descriptor is freed.
- The last modification time for the file is updated.
- All locks created by **fcntl( )** for the file are released.
- If the link count of the file is 0 (zero), the space occupied by the file is freed, and the file is no longer accessible.
- If the file is a socket, the socket is destroyed.
- If the file is a pipe or FIFO, any data remaining in the pipe or FIFO is discarded.

### Use From a Threaded Application

If a thread calls the thread-aware **close( )** to close a file that already has a file operation in progress by a different thread, the new thread is blocked until the prior file operation completes.

## NOTES

To use the **close( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_closez(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occur, the **close( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not a valid open file descriptor.

[EIO] An input or output error occurred. The device that the file is stored on might be in the down state, or both processors that provide access to the device might have failed.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

**RELATED INFORMATION**

Functions: **exec(2)**, **fcntl(2)**, **getsockopt(2)**, **open(2)**, **pipe(2)**, **setsockopt(2)**, **socket(2)**, **spt_closez(2)**, **tdm_execve(2)**, **tdm_execvep(2)**.

Files: **signal(4)**.

**STANDARDS CONFORMANCE**

This function does not return the **errno** value [EINTR].

For an **AF_INET** or **AF_INET6** socket, even if all these are true:

- The socket is connection-oriented.
- The **SO_LINGER** option is enabled for the socket.

- The socket has untransmitted data.

the **close( )** function does not block. The system attempts to deliver unsent data after the **close( )** function is called, although that action can be disabled. See the **setsockopt(2)** reference page for additional information.

HP extensions to the XPG4 Version 2 specification are:

- The **close( )** function can return the **errno** value [EISGUARDIAN].

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**connect -** Connects a socket

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int connect(
        int socket,
        const struct sockaddr *address,
        socklen_t address_len);
```

## PARAMETERS

*socket* Specifies the file descriptor for the socket.

*address* Points to a **sockaddr** structure that contains the address of the peer socket. The length and format of the address depend on the address family of the socket.

For **AF_INET** sockets, a pointer to the address structure **sockaddr_in** must be cast as a **struct sockaddr**. For **AF_INET6** sockets, a pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**. For **AF_UNIX** sockets, a pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*address_len* Specifies the length of the **sockaddr** structure pointed to by the *address* parameter.

## DESCRIPTION

The **connect( )** function requests that a connection be made on a socket. In systems running **AF_UNIX** Release 2 software, both sockets must use the same mode (compatibility or portability). For more information about **AF_UNIX** Release 2, see the *Open System Services Programmer's Guide*.

The **connect( )** function performs a different action for each of the following types of initiating sockets:

- If the initiating socket is not connection-oriented (has the type **SOCK_DGRAM**), then the **connect( )** function sets the peer address but no connection is made. The peer address identifies the socket where all datagrams are sent by subsequent calls to the **send( )** function, and limits the remote sender for subsequent **recv( )** function calls. Datagram sockets can use the **connect( )** function multiple times to communicate with different peers.

  If the socket is a datagram socket and *address* is a null address for the protocol, the address for the peer socket is reset.

- If the initiating socket is connection-oriented (has the type **SOCK_STREAM**), then the **connect( )** function attempts to make a connection to the socket specified by the *address* parameter. Sockets of type **SOCK_STREAM** can successfully connect only once.

When a connection cannot be created immediately and **O_NONBLOCK** is not set for the file descriptor of the socket, the **connect( )** call blocks until one of the following occurs:

- A connection is established
- A timeout occurs
- A signal is caught

If timeout occurs, the **connect( )** call fails and **errno** is set to [ETIMEDOUT]; the connection is aborted.

If a **connect( )** call is interrupted by a signal that is caught while the call is blocked waiting to establish a connection, the **connect( )** call fails and sets **errno** to [EINTR]; the connection is not aborted and is later established asynchronously.

When a connection cannot be created immediately and **O_NONBLOCK** is set for the file descriptor of the socket, the **connect( )** call fails and sets **errno** to [EINPROGRESS]; the connection is not aborted and is later established asynchronously. Subsequent calls to the **connect( )** function for the same socket before the connection is completed will fail and set **errno** to [EALREADY].

**NOTES**

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When an asynchronous connection is complete, a call to the **select( )** function indicates that the file descriptor for the socket is ready for writing.

To use the **connect( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_connectx(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **connect( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **connect( )** function sets **errno** to the corresponding value:

[EACCES] The socket is in the **AF_UNIX** domain and either search permission is denied for a component of the pathname in the **sockaddr** structure, or write access to the specified socket is denied.

[EADDRINUSE]
An attempt was made to establish a connection using addresses that are already in use.

[EADDRNOTAVAIL]
The specified address is not available from this HP NonStop node.

[EAFNOSUPPORT]
Addresses in the specified address family cannot be used with this socket.

[EALREADY] A connection request is already in progress for the specified socket.

[EBADF] The *socket* parameter is not a valid file descriptor.

This error is also returned if the **connect( )** function is thread-aware and the socket becomes invalid (is closed by another thread).

[ECONNREFUSED]
One of these conditions occured:

- The specified address is not listening for connections or rejected the attempt to connect.
- The socket bound to the **AF_UNIX** *address* is not using the same transport provider as the *socket*. This condition can occur if the system is running **AF_UNIX** Release 2 software and the socket bound to *address* is not of the same mode as *socket*.
- For **AF_UNIX** Release 1 socket or an **AF_UNIX** Release 2 socket in compatibility mode:
  - — The caller attempted to connect a socket that previously had been called by the **listen( )** function with a *backlog* parameter less than or equal to 0 (zero), and
  - — There is no pending **accept( )** call to that socket.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EHOSTUNREACH]
The destination host cannot be reached.

[EINPROGRESS]
The socket is marked nonblocking (**O_NONBLOCK** is set) and the requested connection is not yet completed. The connection will be completed asynchronously.

[EINTR] The attempt to connect was interrupted by delivery of a signal. The connection will be completed asynchronously.

This error is also returned if the **connect )** function is thread-aware and a signal received from the **pthread_kill( )** function is not blocked, ignored, or handled.

[EINVAL] One of the following conditions exists:

- The size specified for the *address_len* parameter is not valid for an address in the address family that is used by this connection.
- The **sockaddr** structure contains an invalid address family.

[EIO] The socket is in the **AF_UNIX** domain and an I/O error occurred during a read or write to the file system.

[EISCONN] The specified socket is connection-oriented and is already connected.

[ELOOP] The socket is in the **AF_UNIX** domain and too many symbolic links were encountered in translating the pathname in the **sockaddr** structure.

[ENAMETOOLONG]
The socket is in the **AF_UNIX** domain and one of the following conditions exists:

- The pathname in the **sockaddr** structure exceeds **PATH_MAX** characters.
- A component of the pathname in the **sockaddr** structure exceeds **NAME_MAX** characters.
- The intermediate result of pathname resolution when a symbolic link is part of the pathname in the **sockaddr** structure exceeds **PATH_MAX** characters.

The **pathconf( )** function can be called to obtain the applicable limits.

[ENETDOWN]
The local interface used to reach the destination is down.

[ENETUNREACH]
No route to the network or host is present.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time may succeed.

[ENOENT] The socket is in the **AF_UNIX** domain and one of the following conditions exists:

- A component of the pathname specified in the **sockaddr** structure does not name an existing file.
- The **sockaddr** structure specifies an empty string as a pathname.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOTDIR] The socket is in the **AF_UNIX** domain and a component of the pathname specified in the **sockaddr** structure is not a directory.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EPROTOTYPE]
The specified address has a different type than that of the socket bound to the specified peer address.

[ETIMEDOUT]
The attempt to connect timed out during connection establishment.

[EWOULDBLOCK]
The socket is marked nonblocking and the connection cannot be immediately completed. The application program can select the socket for writing during the connection process. The connection request will take place asynchronously.

## RELATED INFORMATION

Functions: **accept(2)**, **bind(2)**, **getsockname(2)**, **listen(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **socket(2)**, **spt_connectx(2)**.

## STANDARDS CONFORMANCE

The XPG4 specification allows certain behavior to be implementer-defined. The following are choices of the HP implementation:

- The HP implementation does not return the **errno** values [ENOSR] or [EOPNOTSUPP].

The following are HP extensions to the XPG4 specification:

- The **errno** value [ECONNRESET] can be returned when the transport-provider process or OSS Local Server 2 process is unavailable.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**creat -** Creates a regular file in the OSS environment or rewrites an existing file

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <sys/stat.h>      /* optional except for POSIX.1 */
#include <fcntl.h>

int creat(
        const char *path,
        mode_t mode);
```

## PARAMETERS

*path* Points to the pathname of the file to be created or opened for writing.

You cannot specify **/lost+found**, **/dev**, **/dev/tty**, and **/dev/null** for this parameter. Attempts to create these files cause the function call to fail with **errno** set to [EINVAL].

If the *path* parameter refers to a symbolic link, the **creat( )** function opens the file pointed to by the symbolic link.

*mode* Specifies the read, write, and execute permissions of the file and the file type flags for the file.

If the file already exists, you must specify a valid value for this parameter, but this parameter has no effect on the file (you cannot use this parameter to change the permissions of the file).

The value of this parameter is constructed by logically ORing flags that are defined in the **sys/stat.h** header file. If the parent directory of the created file does not have default OSS access control list (ACL) entries, the permissions for the new file are the bit-wise AND of this *mode* parameter with the complement of the process umask (see the **umask(2)** reference page). If the parent directory of the created file has default ACL entries, the permissions for the new file are affected by the value of this parameter but depend on both the support for OSS ACLs on the system on which this process is running and on the fileset that contains the new directory. See "ACL Inheritance" in the **acl(5)** reference page.

The file type flags are described in **DESCRIPTION**.

## DESCRIPTION

This function can create:

- OSS files up to a size limit of approximately 2 gigabytes
- Guardian Format 1 files up to a size limit of approximately 2 gigabytes

For information about creating larger files, see the **creat64(2)** reference page.

The **creat( )** function establishes a connection between the file indicated by the *path* parameter and the returned file descriptor. Subsequent I/O function calls, such as **read( )** and **write( )**, use the opened file descriptor to access that file.

The returned file descriptor is the lowest-numbered file descriptor not currently open for that process. A corresponding Guardian environment file number is also assigned.

The file offset, marking the current position within the file, is set to the beginning of the file. The new file descriptor is set to remain open across the processing of any of the **exec** or **tdm_exec** set of functions. (See the **fcntl(2)** reference page.)

A call to the **creat( )** function is equivalent to this call:

**open(** *path***, O_WRONLY | O_CREAT | O_TRUNC,** *mode* **);**

You cannot use the **creat( )** function to create a first-in, first-out (FIFO) special file. Use the **mkfifo( )** function instead.

If the file does not exist, a regular file is created with these characteristics:

- The owner ID of the file is set to the effective user ID of the process.
- The group ID of the file is determined by the value of the **S_ISGID** flag in the parent directory. If **S_ISGID** is set, the group ID of the file is set to the group ID of the parent directory; otherwise, the group ID of the file is set to the effective group ID of the calling process. If the file is a Guardian file (that is, in the **/G** file system), the group ID is set to that of the primary group of the effective user ID.
- If ACLs are supported, ACL entries are added to the file ACL as described in "ACL Inheritance" in the **acl(5)** reference page.
- The attribute bits and file permission bits are set to the value of the *mode* parameter, modified as listed:
    - File permission bits are set as described in "ACL Inheritance" in the **acl(5)** reference page.
    - The set user ID attribute (**S_ISUID** bit) is cleared.
    - The set group ID attribute (**S_ISGID** bit) is cleared.

  If bits other than the file permission and appropriate file-type flags are set in the *mode* parameter, **errno** is set to [EINVAL].

If the file exists and is a regular file that is successfully opened:

- The length of the file is truncated to 0 (zero).
- The owner and group of the file are unchanged.
- The set user ID attribute of the file mode is cleared.

The open fails if any of these conditions is true:

- The file supports enforced record locks, and another process has locked a portion of the file.
- The file does not allow write access.

### File Type Flags

The file type flags that can be logically ORed into the value specified in the *mode* parameter are:

**S_IFREG** Regular file in the OSS file system or in **/G**, the Guardian file system.

**S_ISVTX** Sticky bit; used only for directories (cannot be used for files in **/G**, the Guardian file system).

### Access Control Lists (ACLs)

The **creat( )** function does not change the ACL for an existing file. For more information about ACLs for existing files, see the **acl(2)**, **chmod(2)**, and **acl(5)** reference pages.

For more information about ACLs, including ACL inheritance for newly created files, see the **acl(5)** reference page.

### Opening Guardian Files

If the file is a Guardian file (that is, a file in the **/G** file system), these rules apply:

- The file can be opened only if it is one of these:

  — An odd, unstructured Enscribe file. In this case, it is opened as a regular file with a primary and secondary extent size that is a multiple of 2. If the extent size is odd, the open fails.

    If the unstructured buffer size was not 4096, a successful open makes the buffer size 4096 (as if the Guardian procedure SETMODE was called for mode 93 with a parameter value of 4096).

  — An EDIT file (file code 101). In this case, it is opened as a regular file for read-only access.

  — A tty simulation process.

  An attempt to open any file (or device) of any other type fails, and **errno** is set to the value of [EINVAL].

  An attempt to open any file on a logical disk volume (virtual disk) administered through the Storage Management Foundation (SMF) fails, and **errno** is set to the value of [ENOTSUP].

  An attempt to open a volume, subvolume, or process (**/G/***vol*, **/G/***vol*/*subvol*, or **/G/***process*, respectively) fails, and **errno** is set to the value of [EISDIR].

- An attempt to open a subvolume with a reserved name beginning with "ZYQ" (for example, **/G/vol2/zyq00004**) fails, and **errno** is set to the value of [EACCES].

- An attempt to open a file within a subvolume with a reserved name beginning with "ZYQ" (for example, **/G/vol2/zyq00004/z000002x**) fails, and **errno** is set to the value of [EACCES].

- If the file is not an EDIT file (that is, the file code is not 101), it is opened in shared exclusion mode.

- If the file is an EDIT file and read-only access is specified, the file is opened in protected exclusion mode in the Guardian environment.

- If the file is an EDIT file and write access is specified, the call fails, and **errno** is set to the value [EINVAL].
- The maximum number of opens is reported by the **sysconf( )** function as the upper limit of opens per process. The actual limit depends on other factors, such as the size of the process file segment (PFS) and the number of existing opens on directories or on files in the Guardian environment.
- When the Guardian file id created, it will be Format 1, odd, unstructured, and file code 180.
- The file is given access permissions compatible with the standard security permissions for the Guardian creator access ID (CAID) of the calling process.

During **creat( )** function processing, all access permissions are checked. This includes Guardian environment checks by Guardian standard security mechanisms (and by the Safeguard product) for Guardian disk file and process access.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted by the same file permissions and owner privileges as any other user ID: It has no special privileges unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

Executable files that have the PRIVSOARFOPEN privilege and that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use From the Guardian Environment

A call to the **creat( )** function in the Guardian environment requires an OSS pathname and returns an OSS file-system file descriptor, regardless of the file system containing the file.

The **creat( )** function belongs to a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file-system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **creat( )** function returns the file descriptor, a nonnegative integer. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the function sets **errno** to the corresponding value:

[EACCES] One of these conditions exists:

- Search permission is denied on a component of the pathname prefix.
- The file does not exist, and write permission is denied for the parent directory.
- The process attempted to open a Guardian subvolume with a reserved name beginning with "ZYQ" or a file within such a subvolume.
- The process attempted to open a static Telserv window that is not yet connected.

[EFAULT] The *path* parameter is an invalid address.

[EFILEBAD] One of these conditions exists:

- The function attempted to open a Guardian EDIT file, but the structure of the file is bad.
- The function attempted to open a Guardian EDIT file, but the corrupted flag is set in the file label.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EGUARDIANOPEN]
The function attempted to open a Guardian EDIT file for write access or for Guardian shared or exclusive exclusion access, but the file has already been opened with a Guardian procedure call.

[EINTR] A signal was caught during the open operation. This value is returned only for character special files (terminal devices) and for FIFO special files.

[EINVAL] One of these conditions exists:

- The call attempted to create a directory named **lost+found** in the root directory of an OSS fileset, or it attempted to create a directory named **/dev**, **/dev/tty**, or **/dev/null** in the root directory of the OSS file system.
- The function call did not specify the *mode* parameter.
- Bits other than the file permission and appropriate file-type flags are set in the *mode* parameter.
- The function attempted to create a Guardian file (that is, a file in the **/G** file system), but the pathname cannot be mapped to a valid Guardian disk file name.
- The function attempted to open a Guardian file (that is, a file in the **/G** file system) of a type other than those permitted.
- The function attempted to create a Guardian temporary file.

[EIO] A physical input or output error occurred. Data might have been lost during transfer.

[EISDIR] One of these conditions exists:

- The named file is an OSS directory.
- The named file is a Guardian directory (**/G** or a directory in the **/G** file system).

[ELOOP] Too many symbolic links were encountered in translating the *path* parameter.

[EMFILE] The system limit for open file descriptors per process has reached the maximum permitted.

[ENAMETOOLONG]
One of these is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENFILE] One of these conditions exists:

- The maximum number of file descriptors of this file type (socket, pipe, etc.) for this processor are already open.
- The limit for open file descriptors of this file type has not been exceeded, but the maximum number of all file descriptors for this processor are already open.

[ENOENT] One of these conditions exists:

- The pathname prefix does not exist.
- The *path* parameter points to an empty string.
- The function attempted to open a file in the Guardian file system, but the specified pathname cannot be mapped to a valid Guardian filename.
- The *path* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOROOT] The root fileset (fileset 0) is not in the STARTED state.

[ENOSPC] The directory that would contain the new file cannot be extended, and the file does not exist.

[ENOTDIR] A component of the pathname prefix is not a directory.

[ENOTSUP] The *path* parameter specifies a Guardian file on an SMF logical volume and one of the following conditions exists:

- The local system is running an RVU prior to J06.15 or H06.26.
- The *path* parameter specifies a file in /E and the remote system is running an RVU prior to J06.15 or H06.26.

[ENXIO] One of these conditions exists:

- The named file is a character special file, and the device associated with this special file does not exist.
- The fileset containing the client's current working directory or root directory is not mounted.

[EOPNOTSUPP]
The named file is a socket bound to the file system (not an AF_INET socket) and cannot be opened.

[EOSSNOTRUNNING]
A required system process is not running.

[EOVERFLOW]
The named file already exists, and the file offset is larger than approximately 2 gigabytes.

[EPERM] One of these conditions exists:

- The call attempted to create a file named **lost+found** in the root directory of an OSS fileset.
- The call attempted to create a file in the **/E** directory.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The named file resides on a read-only fileset, and write access is required.

[ETXTBSY] The file is being executed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **close(2)**, **creat64(2)**, **fcntl(2)**, **lseek(2)**, **lseek64(2)**, **mknod(2)**, **open(2)**, **open64(2)**, **read(2)**, **stat(2)**, **umask(2)**, **write(2)**.

Miscellaneous topics: **acl(5)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- The group ID of the new file is determined by the value of the **O_ISGID** flag in the parent directory.
- If bits other than the file permission and appropriate file-type flags are set in the *mode* parameter, **errno** is set to [EINVAL].
- The **O_TRUNC** flag is ignored for files other than regular files.
- An attempt to open an OSS directory with **creat( )** fails.

HP extensions to the XPG4 Version 2 specification are:

- Opening Guardian files (that is, files in the **/G** file system) is supported, as described under **Opening Guardian Files** in **DESCRIPTION**.
- The **errno** values [EFAULT], [EFILEBAD], [EFSBAD], [EGUARDIANOPEN], [EIO], [ELOOP], [ENOTSUP], [EOSSNOTRUNNING], and [EPERM] can be returned.

## NAME

**creat64 -** Creates a regular file in the OSS environment or rewrites an existing file

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat64(
        const char *path,
        mode_t mode);
```

## PARAMETERS

*path*
: Points to the pathname of the file to be created or opened for writing.

You cannot specify **/lost+found**, **/dev**, **/dev/tty**, and **/dev/null** for this parameter. Attempts to create these files cause the function call to fail with **errno** set to [EINVAL].

If the *path* parameter refers to a symbolic link, the **creat64( )** function opens the file pointed to by the symbolic link.

*mode*
: Specifies the read, write, and execute permissions of the file and the file type flags for the file.

If the file already exists, you must specify a valid value for this parameter, but this parameter has no effect on the file (you cannot use this parameter to change the permissions of the file).

The value of this parameter is constructed by logically ORing flags that are defined in the **sys/stat.h** header file. If the parent directory of the created file does not have default OSS access control list (ACL) entries, the permissions for the new file are the bit-wise AND of this *mode* parameter with the complement of the process umask (see the **umask(2)** reference page). If the parent directory of the created file has default ACL entries, the permissions for the new file are affected by the value of this parameter but depend on both the support for OSS ACLs on the system on which this process is running and on the fileset that contains the new directory. See "ACL Inheritance" in the **acl(5)** reference page.

The file type flags are described in **DESCRIPTION**.

## DESCRIPTION

The **creat64( )** function is similar to the **creat( )** function except that, in addition to supporting smaller files, the **creat64( )** function supports:

- OSS files larger than approximately 2 gigabytes, up to a limit of approximately 1 terabyte (constrained by the space available on the disk volume)
- Both Guardian Format 1 and Guardian Format 2 files, up to the limit described in the *Open System Services Management and Operations Guide*

An application can explicitly call this function when you use the **#define _LARGEFILE64_SOURCE 1** feature test macro or an equivalent compiler command option to compile the application.

An application call to **creat( )** is automatically mapped to this function when you use the **#define _FILE_OFFSET_BITS 64** feature test macro or an equivalent compiler command option to compile the application.

The **creat64( )** function establishes a connection between the file indicated by the *path* parameter and the returned file descriptor. Subsequent I/O function calls, such as **read( )** and **write( )**, use the opened file descriptor to access that file.

The returned file descriptor is the lowest-numbered file descriptor not currently open for that process. A corresponding Guardian environment file number is also assigned.

The file offset, marking the current position within the file, is set to the beginning of the file. The new file descriptor is set to remain open across the processing of any of the **exec** or **tdm_exec** set of functions. (See the **fcntl(2)** reference page.)

A call to the **creat64( )** function is equivalent to this call:

**open64(** *path*, **O_WRONLY | O_CREAT | O_TRUNC,** *mode* **);**

You cannot use the **creat( )** function to create a first-in, first-out (FIFO) special file. Use the **mkfifo( )** function instead.

If the file does not exist, a regular file is created with these characteristics:

- The owner ID of the file is set to the effective user ID of the process.
- The group ID of the file is determined by the value of the **S_ISGID** flag in the parent directory. If **S_ISGID** is set, the group ID of the file is set to the group ID of the parent directory; otherwise, the group ID of the file is set to the effective group ID of the calling process. If the file is a Guardian file (that is, in the **/G** file system), the group ID is set to that of the primary group of the effective user ID.
- If ACLs are supported, ACL entries are added to the file ACL as described in "ACL Inheritance" in the **acl(5)** reference page.
- The file permission and attribute bits are set to the value of the *mode* parameter, modified as listed:
  - The file permission bits are set as described in "ACL Inheritance" in the **acl(5)** reference page.
  - The set user ID attribute (**S_ISUID** bit) is cleared.
  - The set group ID attribute (**S_ISGID** bit) is cleared.

  If bits other than the file permission and appropriate file-type flags are set in the *mode* parameter, **errno** is set to [EINVAL].

If the file exists and is a regular file that is successfully opened:

- The length of the file is truncated to 0 (zero).
- The owner and group of the file are unchanged.
- The set user ID attribute of the file mode is cleared.

The open fails if any of these conditions is true:

- The file supports enforced record locks, and another process has locked a portion of the file.

- The file does not allow write access.

**File Type Flags**

The file type flags that can be logically ORed into the value specified in the *mode* parameter are:

**S_IFREG** Regular file in the OSS file system or in **/G**, the Guardian file system.

**S_ISVTX** Sticky bit; used only for directories (cannot be used for files in **/G**, the Guardian file system).

**Access Control Lists (ACLs)**

The **creat64( )** function does not change the ACL for an existing file. For more information about ACLs for existing files, see the **acl(2)**, **chmod(2)**, and **acl(5)** reference pages.

For more information about ACLs, including ACL inheritance for newly created files, see the **acl(5)** reference page.

**Accessing Files in Restricted-Access Filesets**

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted by the same file permissions and owner privileges as any other user ID: It has no special privileges unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

Executable files that have the PRIVSOARFOPEN privilege and that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

**Opening Guardian Files**

If the file is a Guardian file (that is, a file in the **/G** file system), these rules apply:

- The file can be opened only if it is one of these:

  — An odd, unstructured Enscribe file. In this case, it is opened as a regular file with a primary and secondary extent size that is a multiple of 2. If the extent size is odd, the open fails.

    If the unstructured buffer size was not 4096, a successful open makes the buffer size 4096 (as if the Guardian procedure SETMODE was called for mode 93 with a parameter value of 4096).

  — An EDIT file (file code 101). In this case, it is opened as a regular file for read-only access.

  — A tty simulation process.

  An attempt to open any file (or device) of any other type fails, and **errno** is set to the value of [EINVAL].

  An attempt to open any file on a logical disk volume (virtual disk) administered through the Storage Management Foundation (SMF) fails, and **errno** is set to the value of [ENOTSUP].

  An attempt to open a volume, subvolume, or process (**/G/***vol*, **/G/***vol*/*subvol*, or **/G/***process*, respectively) fails, and **errno** is set to the value of [EISDIR].

- An attempt to open a subvolume with a reserved name beginning with "ZYQ" (for example, **/G/vol2/zyq00004**) fails, and **errno** is set to the value of [EACCES].
- An attempt to open a file within a subvolume with a reserved name beginning with "ZYQ" (for example, **/G/vol2/zyq00004/z000002x**) fails, and **errno** is set to the value of [EACCES].
- If the file is not an EDIT file (that is, the file code is not 101), it is opened in shared exclusion mode.
- If the file is an EDIT file and read-only access is specified, the file is opened in protected exclusion mode in the Guardian environment.
- If the file is an EDIT file and write access is specified, the call fails, and **errno** is set to the value [EINVAL].
- The maximum number of opens is reported by the **sysconf( )** function as the upper limit of opens per process. The actual limit depends on other factors, such as the size of the process file segment (PFS) and the number of existing opens on directories or on files in the Guardian environment.
- When a Guardian file is created, the file will be Format 2, odd, unstructured, and file code 180.
- If the open causes file creation, the file is given access permissions compatible with the standard security permissions for the Guardian creator access ID (CAID) of the calling process.

During **creat64( )** function processing, all access permissions are checked. This includes Guardian environment checks by Guardian standard security mechanisms (and by the Safeguard product) for Guardian disk file and process access.

### Use From the Guardian Environment

A call to the **creat64( )** function in the Guardian environment requires an OSS pathname and returns an OSS file-system file descriptor, regardless of the file system containing the file.

The **creat64( )** function belongs to a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file-system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **creat64( )** function returns the file descriptor, a nonnegative integer. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the function sets **errno** to the corresponding value:

[EACCES] One of these conditions exists:

- Search permission is denied on a component of the pathname prefix.
- The file does not exist, and write permission is denied for the parent directory.
- The process attempted to open a Guardian subvolume with a reserved name beginning with "ZYQ" or a file within such a subvolume.
- The process attempted to open a static Telserv window that is not yet connected.

[EFAULT] The *path* parameter points to a location outide of the allocated address space of the process.

[EFILEBAD] One of these conditions exists:

- The function attempted to open a Guardian EDIT file, but the structure of the file is bad.
- The function attempted to open a Guardian EDIT file, but the corrupted flag is set in the file label.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EGUARDIANOPEN]
The function attempted to open a Guardian EDIT file for write access or for Guardian shared or exclusive exclusion access, but the file has already been opened with a Guardian procedure call.

[EINTR] A signal was caught during the open operation. This value is returned only for character special files (terminal devices) and for FIFO special files.

[EINVAL] One of these conditions exists:

- The call attempted to create a directory named **lost+found** in the root directory of an OSS fileset, or it attempted to create a directory named **/dev**, **/dev/tty**, or **/dev/null** in the root directory of the OSS file system.
- The function call did not specify the *mode* parameter.
- Bits other than the file permission and appropriate file-type flags are set in the *mode* parameter.
- The function attempted to create a Guardian file (that is, a file in the **/G** file system), but the pathname cannot be mapped to a valid Guardian disk file name.
- The function attempted to open a Guardian file (that is, a file in the **/G** file system) of a type other than those permitted.

- The function attempted to create a Guardian temporary file.

[EIO] A physical input or output error occurred. Data might have been lost during transfer.

[EISDIR] One of these conditions exists:

- The named file is an OSS directory.
- The named file is a Guardian directory (**/G** or a directory in the **/G** file system).

[ELOOP] Too many symbolic links were encountered in translating the *path* parameter.

[EMFILE] The system limit for open file descriptors per process has reached the maximum permitted.

[ENAMETOOLONG]
One of these is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENFILE] One of these conditions exists:

- The maximum number of file descriptors of this file type (socket, pipe, etc.) for this processor are already open.
- The limit for open file descriptors of this file type has not been exceeded, but the maximum number of all file descriptors for this processor are already open.

[ENOENT] One of these conditions exists:

- The pathname prefix does not exist.
- The *path* parameter points to an empty string.
- The function attempted to open a file in the Guardian file system, but the specified pathname cannot be mapped to a valid Guardian filename.
- The *path* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOROOT] The root fileset (fileset 0) is not in the STARTED state.

[ENOSPC] The directory that would contain the new file cannot be extended, and the file does not exist.

[ENOTDIR] A component of the pathname prefix is not a directory.

[ENOTSUP] The *path* parameter specifies a Guardian file on an SMF logical volume and one of the following conditions exists:

- The local system is running an RVU prior to J06.15 or H06.26.
- The *path* parameter specifies a file in /E and the remote system is running an RVU prior to J06.15 or H06.26.

[ENXIO] One of these conditions exists:

- The named file is a character special file, and the device associated with this special file does not exist.
- The fileset containing the client's current working directory or root directory is not mounted.

[EOPNOTSUPP]
The named file is a socket bound to the file system (not an AF_INET socket) and cannot be opened.

[EOSSNOTRUNNING]
A required system process is not running.

[EPERM] One of these conditions exists:

- The call attempted to create a file named **lost+found** in the root directory of an OSS fileset.
- The call attempted to create a file in the **/E** directory.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The named file resides on a read-only fileset, and write access is required.

[ETXTBSY] The file is being executed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **close(2)**, **creat(2)**, **fcntl(2)**, **lseek(2)**, **lseek64(2)**, **mknod(2)**, **open(2)**, **open64(2)**, **read(2)**, **stat(2)**, **stat64(2)**, **umask(2)**, **write(2)**.

Miscellaneous topics: **acl(5)**.

## STANDARDS CONFORMANCE

This function is an HP extension to the XPG4 Version 2 specification.

## NAME

**dup -** Duplicates an open file descriptor

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

**#include <unistd.h>**

**int dup(**
    **int** *filedes***);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

## DESCRIPTION

The **dup( )** function returns a new file descriptor for the open file specified by the *filedes* parameter. This file descriptor:

- Is the lowest-numbered available file descriptor
- References the same open
- Returns the same file pointer as the original file (that is, both file descriptors share one file pointer if the object is a file)
- Returns the same access mode (read, write, or read/write)
- Returns the same file status flags (that is, both file descriptors share the same file status flags)
- Clears the close-on-exec flag (**FD_CLOEXEC** bit) associated with the new file descriptor so that the file remains open across calls to any function in the **exec**, **tdm_exec**, and **tdm_spawn** sets of functions

## NOTES

The **dup( )** function provides an alternative interface to the service provided by the **fcntl( )** function by using the **F_DUPFD** value of the *request* parameter. The call:

```
fid = dup( file1 );
```

is equivalent to:

```
fid = fcntl( file1, F_DUPFD, 0 );
```

## RETURN VALUES

Upon successful completion, the **dup( )** function returns a new file descriptor. If the **dup( )** function fails, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **dup( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not a valid open file descriptor.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[EMFILE] The number of file descriptors exceeds the maximum number of opens permitted.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation on an input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **close(2)**, **dup2(2)**, **exec(2)**, **fcntl(2)**, **open(2)**, **read(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **write(2)**.

## STANDARDS CONFORMANCE

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EISGUARDIAN] and [EWRONGID] can be returned.

## NAME

**dup2 -** Duplicates and controls an open file descriptor

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <unistd.h>

int dup2(
        int filedes,
        int new);
```

## PARAMETERS

*filedes*
Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function, or the thread-aware **accept( )**, **dup2( )**, or **fcntl( )** function.

*new*
Specifies the open file descriptor that is returned by the **dup2( )** function. If this descriptor is already in use, it is first deallocated as if it had been closed.

## DESCRIPTION

The **dup2( )** function returns a new file descriptor on the open file specified by the *filedes* parameter. If *new* is less than 0 (zero) or greater than or equal to the maximum number of opens permitted, **dup2( )** returns -1 with **errno** set to [EBADF].

The new file descriptor:

- Is the value specified as the *new* parameter:
  - If *filedes* is a valid file descriptor and is equal to *new*, **dup2( )** returns *new* without closing it.
  - If *filedes* is not a valid file descriptor, **dup2( )** returns -1 and does not close *new*.
  - The value returned is equal to the value of *new* upon successful completion, or it is -1 upon failure.
- References the same open.
- Returns the same file pointer as the original file (that is, both file descriptors share one file pointer if the object is a file).
- Returns the same access mode (read, write, or read/write).

- Returns the same file status flags (that is, both file descriptors share the same file status flags).
- Clears the close-on-exec flag (**FD_CLOEXEC** bit) associated with the new file descriptor so that the file remains open across calls to any function in the **exec**, **tdm_exec**, and **tdm_spawn** sets of functions.

## NOTES

The **dup2( )** function provides an alternative interface to the service provided by the **fcntl( )** function by using the **F_DUPFD** value of the *request* parameter. The call:

```
fid = dup2( file1, file2 );
```

is equivalent to:

```
close( file2 );
fid = fcntl( file1, F_DUPFD, file2 );
```

To use the **dup2( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_dup2x(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the **dup2( )** function returns a new file descriptor. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **dup2( )** function sets **errno** to the corresponding value:

[EBADF] One of these conditions exists:

- The *filedes* parameter is not a valid open file descriptor.
- The *new* parameter file descriptor is negative or greater than the maximum number of opens permitted.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation on an input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **close(2)**, **dup(2)**, **exec(2)**, **fcntl(2)**, **open(2)**, **read(2)**, **spt_dup2x(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **write(2)**.

## STANDARDS CONFORMANCE

The **dup2( )** function does not return the **errno** value [EINTR].

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EISGUARDIAN] and [EWRONGID] can be returned.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

# Section 2. System Functions (e)

This section contains reference pages for Open System Services (OSS) system function calls with names that begin with **e**. These reference pages reside in the **cat2** directory and are sorted alphabetically by U.S. English conventions in this section.

## NAME

**exec -** Specifies a set of functions that execute a file

## DESCRIPTION

The **exec** set of functions (**execl( )**, **execle( )**, **execlp( )**, **execv( )**, **execve( )**, and **execvp( )**) replace the current process image with a new process image. The new image is constructed from a regular executable file, called a new process image file. The new process image file is formatted as an executable text or binary file in one of the formats recognized by the **exec** set of functions.

A successful call to any function in the **exec** set of functions does not return, because the calling process image is overlaid by the new process image.

When a program is executed as a result of a call to a function in the **exec** set of functions, it is entered as a function call as follows:

```
int main(
        int argc,
        char *argv[ ],
        char *env[ ]);
```

Here, the *argc* parameter is the argument count, the *argv***[ ]** parameter is an array of character pointers to the arguments themselves, and *env***[ ]** is a pointer to a character array listing the environment variables.

In addition, the following variable is initialized for the new process as a pointer to an array of character pointers to the environment strings:

**extern char ****environ*;

The *argv***[ ]** array is terminated by a null pointer. The null pointer is not counted in *argc*.

The arguments specified by a program with one of the **exec** set of functions are passed on to the new process image in the corresponding arguments to the **main( )** function.

The *env***[ ]** parameter for the main function is an HP extension and is not the preferred method of obtaining the environment variables for the child process. Use of the ****environ** array is the preferred method.

For additional information, refer to the reference page for a specific function in the **exec** set of functions.

## RELATED INFORMATION

Commands: **eld(1)**, **ld(1)**, **nld(1)**.

Functions: **alarm(3)**, **_exit(2)**, **execl(2)**, **execle(2)**, **execlp(2)**, **execv(2)**, **execve(2)**, **execvp(2)**, **fcntl(2)**, **fork(2)**, **getenv(3)**, **putenv(3)**, **semget(2)**, **sigaction(2)**, **system(3)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **times(3)**, **ulimit(3)**, **umask(2)**.

Miscellaneous: **environ(5)**.

**NAME**

**execl -** Executes a file using a pathname, a set of argument strings, and ****environ**

**LIBRARY**

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

**SYNOPSIS**

```
#include <unistd.h>

extern char **environ;

int execl(
        const char *path,
        const char *arg, ...);
```

**PARAMETERS**

****environ**
Points to an array of character pointers to environment strings. The environment strings define the OSS environment for the new process. The **environ** array is terminated by a null pointer.

The ****environ** array of the new process is also passed as the *env***[ ]** array in the call to the **main( )** function of the new process. Refer to **Entering the New Process** later in this reference page.

*path*
Points to a null-terminated string containing a pathname that identifies the new process image file. The pathname is absolute if it starts with a slash (/) character. Otherwise, the pathname is relative and is resolved by prefixing the current working directory.

If the final component of the *path* parameter names a symbolic link, the link is traversed and pathname resolution continues.

*arg*
Points to a null-terminated string containing an argument to be made visible to the main function of the new program. The first such argument should point to the null-terminated string containing the filename of the new process image. The last of these arguments must be a null pointer.

These strings constitute the argument list available to the new process image.

**DESCRIPTION**

The **execl( )** function is one of the **exec** set of functions. The **exec** set of functions replace the current process image with a new process image. The new image is constructed from a regular executable file, called a new process image file. The new process image file is formatted as an executable text or binary file in one of the formats recognized by the **exec** set of functions.

A successful **execl( )** function call does not return, because the calling process image is overlaid by the new process image.

**Entering the New Process**

When a program is executed as a result of a call to a function in the **exec** set of functions, it is entered as a function call as follows:

```
int main(
        int argc,
        char *argv[ ],
        char *env[ ]);
```

Here, the *argc* parameter is the argument count, the *argv***[ ]** parameter is an array of character pointers to the arguments themselves, and *env***[ ]** is a pointer to a character array listing the environment variables. The *argv***[ ]** array is terminated by a null pointer. The null pointer is not counted in *argc*.

The arguments specified by a program with one of the **exec** set of functions are passed on to the new process image in the corresponding arguments to the **main( )** function.

The *envp***[ ]** parameter for the main function is an HP extension and is not the preferred method of obtaining the environment variables for the new process. Use of the ****environ** array is the preferred method.

**Passing the Arguments and the Environment**

The number of bytes available for the new process's combined argument and environment lists has a system-imposed limit. This limit, which includes the pointers and the null terminators on the strings, is available by calling the **sysconf(_SC_ARG_MAX)** function.

**Executing a Binary File**

If the file specified in the function call is a binary executable file, the function loads the file directly.

**Executing a Text File**

If the file specified in the function call is not a binary executable file, the function examines the file to determine whether it is an executable text file. The function checks for a header line in the following format:

**#!** *interpreter_name* [*optional_string*]

The **#!** notation identifies the file as an executable text file. The new process image filename is constructed from the process image filename in the *interpreter_name* string, treating it like the *path* parameter. The arguments passed to the new process are modified as follows:

- The *argv***[0]** parameter is set to the name of the interpreter.
- If the *optional_string* portion is present, *argv***[1]** is set to *optional_string*.
- The next element of *argv***[ ]** is set to the original value of *path*.
- The remaining elements of *argv***[ ]** are set to the the second and subsequent values of the *arg* parameter.
- The first value of *arg* is discarded.

The **S_ISUID** and **S_ISGID** mode bits of an executable text file are honored. Those bits of the *interpreter_name* command interpreter are ignored.

**When the File Is Invalid**

If the process image file is not a valid executable object, or if the text file does not contain the header line, the **execl( )** function call fails and sets **errno** to the value of [ENOEXEC].

**Open Files**

File descriptors open in the calling process image remain open in the new process image, except for those:

- Whose close-on-exec flag **FD_CLOEXEC** is set (see the **fcntl(2)** reference page)
- Opened using a Guardian function or procedure call

If the process file segment of the new process image is smaller than the process file segment of the calling process image and if the calling process image has a large number of file descriptors open, then the system might not be able to propagate all the open file descriptors to the new process image. When this situation occurs, the function call fails and **errno** is set to the value of [EMFILE].

For those file descriptors that remain open, all attributes of the open file descriptor, including file locks, remain unchanged. All directory streams are closed.

**Shared Memory**

Any attached shared memory segments are detached by a successful call to a function in the **exec** set of functions. Refer to the **shmat(2)** reference page for additional information about shared memory segment use.

**Semaphores**

Semaphore set IDs attached to a calling process are also attached to the new process. The new process also inherits the adjust-on-exit (**semadj**) values of the calling process.

Refer to the **semget(2)** reference page for additional information about semaphore use.

**Signals**

Signals set to:

- The default action (**SIG_DFL**) in the calling process image are set to the default action in the new process image.
- Be ignored (**SIG_IGN**) by the calling process image are set to be ignored by the new process image.
- Cause abnormal termination (**SIG_ABORT**) in the calling process image are set to that action in the new process image.
- Cause entry into the debugger (**SIG_DEBUG**) in the calling process image are set to that action in the new process image.
- Be caught by the calling process image are set to the default action in the new process image.

See the **signal(4)** reference page either online or in the *Open System Services System Calls Reference Manual*.

**User ID and Group ID**

If the set-user-ID mode bit of the new process image file is set (see the **chmod(2)** reference page), the effective user ID of the new process image is set to the owner ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved-set-user ID and the saved-set-group ID) for use by the **setuid( )** function.

The **_POSIX_SAVED_IDS** flag is defined **TRUE**.

**OSS Attributes**

The following OSS attributes of the calling process image are unchanged after successful completion of any of the **exec** set of functions:

- OSS process ID (PID)
- Parent process ID
- Process group ID

- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- The time left until an alarm clock signal is posted (see the **alarm(3)** reference page)
- Current working directory
- Root directory
- File mode creation mask (see the **umask(2)** reference page)
- Process signal mask (see the **sigprocmask(2)** reference page)
- Pending signals (see the **sigpending(2)** reference page)
- The **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** fields of the **tms** structure
- File size limit (see the **ulimit(2)** reference page)

Upon successful completion of the function call, the **st_atime** field of the file is marked for update.

The POSIX.1 standard does not specify the effect on the **st_atime** field when the function call fails but does find the file. Likewise, the HP implementation does not guarantee the outcome. Under these circumstances, this field should not be used for further processing.

**Guardian Attributes**

The newly created OSS process retains the following Guardian attributes of the process that calls one of the **exec** set of functions:

- Priority
- Processor on which the process executes
- Home terminal
- Job ID
- DEFINE mode switch
- Process access ID (PAID), unless the **S_ISUID** mode bit of the new image file is set
- Security group list
- Job ancestor or GMOM
- Unread system message index (PCBMCNT)

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Outstanding incoming and outgoing message limits

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.

- Login, remote, and saveabend flags
- File creation mask

The Guardian attributes of the new process differ from those of the calling process in the following ways:

- Segments created or shared using Guardian procedure calls such as SEGMENT_ALLOCATE_ are not inherited.
- The program file is the file specified in the function call.
- The library file is specified in the program file.
- The new process does not inherit the caller's extended swap file (if any). For a G-series TNS process or an accelerated process, the extended data segment is managed by the Kernel Managed Storage Facility (KMSF).
- The process name for the new process is system-generated if the RUNNAMED option is set in the program file. Otherwise the process is unnamed.
- The size of the data segment of the new process is set in the program file
- The remote login flag (PCBREMID) is set to off if the program file has had its **S_ISUID** mode bit set. Otherwise, the remote login flag is set the same as for the caller.
- The size of the extended data segment of the new process is set in the program file
- The DEFINEs inherited by the new process depend on the setting of DEFINE mode in the caller. If DEFINE mode in the caller is ON, all the caller's DEFINEs are inherited. If DEFINE mode is OFF, no DEFINEs are inherited.
- The process identification number (PIN) of the new process is unrelated to that of the calling process. The PIN of the new process is unrestricted if both of the following are true:

  — The HIGHPIN flag is set in the program file and any user library file.

  — The PIN of the calling process was unrestricted.

  If the PIN of the new process is restricted, then the PIN is in the range 0 through 254.
- The creator access ID (CAID) is set to the process access ID (PAID) of the calling process.
- The PAID depends on whether the **S_ISUID** mode bit of the image file is set. If so, the PAID is based on the file owner ID. If not, the PAID is the same as for the caller. (The **S_ISUID** mode bit of the image file has no effect on the security group list.)
- The MOM field for the new process depends on whether the calling process is named. If so, the MOM field for the new process is set to the caller's ANCESTOR field. Otherwise, the MOM field for the new process is set to the caller's MOM field.
- System debugger selection for the new process is based on Inspect mode.
- Code breakpoints and memory breakpoints are not inherited.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Use From the Guardian Environment

If called from a Guardian process, the function call fails and **errno** is set to [ENOTOSS].

## RETURN VALUES

If the **execl( )** function returns to the calling process image, an error has occurred; the return value is -1, and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the function sets **errno** to the corresponding value. For any of these error conditions, file descriptors marked close-on-exec are not closed, signals set to be caught are not set to the default action, and none of the following are changed:

- The value of the global variable **environ**
- The pointers contained within the global variable **environ**
- The elements pointed to by **environ** pointers
- The effective user ID of the current process
- The effective group ID of the current process

[E2BIG] The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit. The limit can be obtained by calling the **sysconf(_SC_ARG_MAX)** function.

[EACCES] One of the following conditions exists:

- Search permission is denied for the directory components of the pathname prefix to the process image file.
- The new process image file, any library file, or script file denies execution permission.
- The new process image file is not a regular file.

[EAGAIN] System resources such as disk space, process control block (PCB) space, MAPPOOL space, stack space, or PFS space are temporarily inadequate.

[EFAULT] An input address parameter is outside valid bounds limits.

[EINVAL] The new process image file is a binary executable file with invalid attributes.

[EIO] Some physical input or output error has occurred. Either a file cannot be opened because of an input or output error, or data has been lost during an input or output transfer. This value is used for errors on the object file of a loaded program or library, or during data transfer with a Guardian environment home terminal.

For systems running J06.07 and later J-series RVUs or H06.18 or later H-series RVUs, this error can also occur when the OSS file system is out of memory and one or more open files cannot be propagated from the parent process to the child process. In this case, if you are running a program from the shell with the shell reporting any errors, you might see an error like this:

/bin/-sh: /bin/ps: tdm_execve(): failed with unexpected error pr_errno=(4005) pr_TPCerror=(110) pr_TPCdetail=(36)

where:

- **pr_errno** is the [EIO] error

- **pr_TPCerror** is the Guardian PROCESS_LAUNCH_ or PROCESS_CREATE_ error.

[ELOOP] Too many symbolic links were encountered in pathname resolution.

[EMFILE] The maximum number of files is open. The process attempted to open more than the maximum number of file descriptors allowed for the process. The process file segment (PFS) of the new process might be smaller than that of the calling process.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the pathname pointed to by the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENODEV] The system cannot find the device containing the fileset containing the process image file.

[ENOENT] One of the following conditions exists:

- One or more components of the new process image file's pathname do not exist.
- The *path* parameter points to an empty string.

[ENOEXEC] The new process image file has the appropriate access permissions, but it is neither in the correct binary executable format nor a valid executable text file.

[ENOMEM] Required resources are not available. Subsequent calls to the same function will not succeed for the same reason.

Possible causes of this error include insufficient primary memory (stack, globals, or heap) for the new process.

[ENOTDIR] A component of the path prefix of the new process image file is not a directory.

[ENOTOSS] The calling process is not an OSS process. A function in the **exec** set of functions cannot be called from the Guardian environment.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[ETXTBSY] The new process image file is currently open for writing by a process.

[EUNKNOWN]
Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

## RELATED INFORMATION

Commands: **eld(1)**, **ld(1)**, **nld(1)**.

Functions: **alarm(3)**, **_exit(2)**, **execle(2)**, **execlp(2)**, **execv(2)**, **execve(2)**, **execvp(2**, **fcntl(2)**, **fork(2)**, **getenv(3)**, **putenv(3)**, **semget(2)**, **sigaction(2)**, **system(3)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **times(3)**, **ulimit(3)**, **umask(2)**.

Miscellaneous: **environ(5)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- Guardian attributes are associated with the new OSS process. See **Guardian Attributes** under **DESCRIPTION**.
- The contents of the **st_atime** field following a failed function call in which the file was found should not be depended upon for further processing.
- The use of ***env[ ]** as a parameter in the call to **main( )** is an HP extension.

The following are HP extensions to the XPG4 Version 2 specification:

- Text files containing the **#!** *interpreter_name* **[***optional_string***]** header line can execute.
- The [EINVAL], [EIO], [ENODEV], [ENOTOSS], and [EUNKNOWN] error values are an HP extension.

## NAME

**execle -** Executes a file using a pathname, a set of argument strings, and an undeclared *envp* array

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/yosskdll**

## SYNOPSIS

```
#include <unistd.h>

extern char **environ;

int execle(
        const char *path,
        const char *arg, ...,
      char *const envp[ ]);
```

## PARAMETERS

****environ**
: Points to an array of character pointers to environment strings. The environment strings define the OSS environment for the calling process. The **environ** array is terminated by a null pointer.

  When the new process is created, the corresponding ****environ** array is not initialized for it with the content of the ****environ** array of the calling process. Instead, the undeclared *envp***[ ]** array that can follow the *arg* parameter list is written into the ****environ** array of the new process when the **execle( )** function call is processed.

  The ****environ** array of the new process is also passed as the *env***[ ]** array in the call to the **main( )** function of the new process. Refer to **Entering the New Process** later in this reference page.

*path*
: Points to a null-terminated string containing a pathname that identifies the new process image file. The pathname is absolute if it starts with a slash (/) character. Otherwise, the pathname is relative and is resolved by prefixing the current working directory.

  If the final component of the *path* parameter names a symbolic link, the link is traversed and pathname resolution continues.

*arg*
: Points to a null-terminated string containing an argument to be made visible to the main function of the new program. The first such argument should point to the null-terminated string containing the filename of the new process image. The last of these arguments must be a null pointer.

  These strings constitute the argument list available to the new process image.

## DESCRIPTION

The **execle( )** function is one of the **exec** set of functions. The **exec** set of functions replace the current process image with a new process image. The new image is constructed from a regular executable file, called a new process image file. The new process image file is formatted as an executable text or binary file in one of the formats recognized by the **exec** set of functions.

A successful **execle( )** function call does not return, because the calling process image is overlaid by the new process image.

**Entering the New Process**

When a program is executed as a result of a call to a function in the **exec** set of functions, it is entered as a function call as follows:

**int main(**
**int** *argc***,**
**char** ∗*argv***[ ],**
**char** ∗*env***[ ]);**

Here, the *argc* parameter is the argument count, the *argv***[ ]** parameter is an array of character pointers to the arguments themselves, and *env***[ ]** is a pointer to a character array listing the environment variables. The *argv***[ ]** array is terminated by a null pointer. The null pointer is not counted in *argc*.

The arguments specified by a program with one of the **exec** set of functions are passed on to the new process image in the corresponding arguments to the **main( )** function.

The *env***[ ]** parameter for the main function is an HP extension and is not the preferred method of obtaining the environment variables for the new process. Use of the the ****environ** array of the new process is the preferred method.

**Passing the Arguments and the Environment**

Instead of passing the ****environ** array of the calling process, the environment for the new process is provided by following the null pointer that terminates the list of *arg* parameters with an additional parameter as if it were declared as:

**char** ∗ **const** *envp***[ ]**

The *envp***[ ]** parameter names an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated with a null pointer.

The number of bytes available for the new process's combined argument and environment lists has a system-imposed limit. This limit, which includes the pointers and the null terminators on the strings, is available by calling the **sysconf(_SC_ARG_MAX)** function.

**Executing a Binary File**

If the file specified in the function call is a binary executable file, the function loads the file directly.

**Executing a Text File**

If the file specified in the function call is not a binary executable file, the function examines the file to determine whether it is an executable text file. The function checks for a header line in the following format:

**#!** *interpreter_name* [*optional_string*]

The **#!** notation identifies the file as an executable text file. The new process image filename is constructed from the process image filename in the *interpreter_name* string, treating it like the *path* parameter. The arguments passed to the new process are modified as follows:

- The *argv***[0]** parameter is set to the name of the interpreter.
- If the *optional_string* portion is present, *argv***[1]** is set to *optional_string*.
- The next element of *argv***[ ]** is set to the original value of *path*.
- The remaining elements of *argv***[ ]** are set to the original elements of *argv***[ ]**, starting with the second and subsequent values of the *arg* parameter.

- The first value of *arg* is discarded.

The **S_ISUID** and **S_ISGID** mode bits of an executable text file are honored. Those bits of the *interpreter_name* command interpreter are ignored.

**When the File Is Invalid**

If the process image file is not a valid executable object, or if the text file does not contain the header line, the **execle( )** function call fails and sets **errno** to the value of [ENOEXEC].

**Open Files**

File descriptors open in the calling process image remain open in the new process image, except for those:

- Whose close-on-exec flag **FD_CLOEXEC** is set (see the **fcntl(2)** reference page)
- Opened using a Guardian function or procedure call

If the process file segment of the new process image is smaller than the process file segment of the calling process image and if the calling process image has a large number of file descriptors open, then the system might not be able to propagate all the open file descriptors to the new process image. When this situation occurs, the function call fails and **errno** is set to the value of [EMFILE].

For those file descriptors that remain open, all attributes of the open file descriptor, including file locks, remain unchanged. All directory streams are closed.

**Shared Memory**

Any attached shared memory segments are detached by a successful call to a function in the **exec** set of functions. Refer to the **shmat(2)** reference page for additional information about shared memory segment use.

**Semaphores**

Semaphore set IDs attached to a calling process are also attached to the new process. The new process also inherits the adjust-on-exit (**semadj**) values of the calling process.

Refer to the **semget(2)** reference page for additional information about semaphore use.

**Signals**

Signals set to:

- The default action (**SIG_DFL**) in the calling process image are set to the default action in the new process image.
- Be ignored (**SIG_IGN**) by the calling process image are set to be ignored by the new process image.
- Cause abnormal termination (**SIG_ABORT**) in the calling process image are set to that action in the new process image.
- Cause entry into the debugger (**SIG_DEBUG**) in the calling process image are set to that action in the new process image.
- Be caught by the calling process image are set to the default action in the new process image.

See the **signal(4)** reference page either online or in the *Open System Services System Calls Reference Manual*.

### User ID and Group ID

If the set-user-ID mode bit of the new process image file is set (see the **chmod(2)** reference page), the effective user ID of the new process image is set to the owner ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved-set-user ID and the saved-set-group ID) for use by the **setuid( )** function.

The **_POSIX_SAVED_IDS** flag is defined **TRUE**.

### OSS Attributes

The following OSS attributes of the calling process image are unchanged after successful completion of any of the **exec** set of functions:

- OSS process ID (PID)
- Parent process ID
- Process group ID
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- The time left until an alarm clock signal is posted (see the **alarm(3)** reference page)
- Current working directory
- Root directory
- File mode creation mask (see the **umask(2)** reference page)
- Process signal mask (see the **sigprocmask(2)** reference page)
- Pending signals (see the **sigpending(2)** reference page)
- The **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** fields of the **tms** structure
- File size limit (see the **ulimit(2)** reference page)

Upon successful completion of the function call, the **st_atime** field of the file is marked for update.

The POSIX.1 standard does not specify the effect on the **st_atime** field when the function call fails but does find the file. Likewise, the HP implementation does not guarantee the outcome. Under these circumstances, this field should not be used for further processing.

### Guardian Attributes

The newly created OSS process retains the following Guardian attributes of the process that calls one of the **exec** set of functions:

- Priority
- Processor on which the process executes

- Home terminal
- Job ID
- DEFINE mode switch
- Process access ID (PAID), unless the **S_ISUID** mode bit of the new image file is set
- Security group list
- Job ancestor or GMOM
- Unread system message index (PCBMCNT)

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Outstanding incoming and outgoing message limits

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Login, remote, and saveabend flags
- File creation mask

The Guardian attributes of the new process differ from those of the calling process in the following ways:

- Segments created or shared using Guardian procedure calls such as SEGMENT_ALLOCATE_ are not inherited.
- The program file is the file specified in the function call.
- The library file is specified in the program file.
- The new process does not inherit the caller's extended swap file (if any). For a G-series TNS process or an accelerated process, the extended data segment is managed by the Kernel Managed Storage Facility (KMSF).
- The process name for the new process is system-generated if the RUNNAMED option is set in the program file. Otherwise the process is unnamed.
- The size of the data segment of the new process is set in the program file.
- The remote login flag (PCBREMID) is set to off if the program file has had its **S_ISUID** mode bit set. Otherwise, the remote login flag is set the same as for the caller.
- The size of the extended data segment of the new process is set in the program file.
- The DEFINEs inherited by the new process depend on the setting of DEFINE mode in the caller. If DEFINE mode in the caller is ON, all the caller's DEFINEs are inherited. If DEFINE mode is OFF, no DEFINEs are inherited.
- The process identification number (PIN) of the new process is unrelated to that of the calling process. The PIN of the new process is unrestricted if both of the following are true:

  — The HIGHPIN flag is set in the program file and any user library file.

  — The PIN of the calling process was unrestricted.

  If the PIN of the new process is restricted, then the PIN is in the range 0 through 254.

- The creator access ID (CAID) is set to the process access ID (PAID) of the calling process.
- The PAID depends on whether the **S_ISUID** mode bit of the image file is set. If so, the PAID is based on the file owner ID. If not, the PAID is the same as for the caller. (The **S_ISUID** mode bit of the image file has no effect on the security group list.)
- The MOM field for the new process depends on whether the calling process is named. If so, the MOM field for the new process is set to the caller's ANCESTOR field. Otherwise, the MOM field for the new process is set to the caller's MOM field.
- System debugger selection for the new process is based on Inspect mode.
- Code breakpoints and memory breakpoints are not inherited.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Use From the Guardian Environment

If called from a Guardian process, the function call fails and **errno** is set to [ENOTOSS].

## RETURN VALUES

If the **execle( )** function returns to the calling process image, an error has occurred; the return value is -1, and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the function sets **errno** to the corresponding value. For any of these error conditions, file descriptors marked close-on-exec are not closed, signals set to be caught are not set to the default action, and none of the following are changed:

- The *envp*[ ] array of pointers
- The elements pointed to by this array
- The effective user ID of the current process
- The effective group ID of the current process

[E2BIG] The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit. The limit can be obtained by calling the **sysconf(_SC_ARG_MAX)** function.

[EACCES] One of the following conditions exists:

- Search permission is denied for the directory components of the pathname prefix to the process image file.
- The new process image file, any library file, or script file denies execution permission.
- The new process image file is not a regular file.

[EAGAIN] System resources such as disk space, process control block (PCB) space, MAPPOOL space, stack space, or PFS space are temporarily inadequate.

[EFAULT] An input address parameter is outside valid bounds limits.

[EINVAL] The new process image file is a binary executable file with invalid attributes.

[EIO] Some physical input or output error has occurred. Either a file cannot be opened because of an input or output error, or data has been lost during an input or output transfer. This value is used for errors on the object file of a loaded program or library, or during data transfer with a Guardian environment home terminal.

For systems running J06.07 and later J-series RVUs or H06.18 or later H-series RVUs, this error can also occur when the OSS file system is out of memory and one or more open files cannot be propagated from the parent process to the child process. In this case, if you are running a program from the shell with the shell reporting any errors, you might see an error like this:

/bin/-sh: /bin/ps: tdm_execve(): failed with unexpected error pr_errno=(4005) pr_TPCerror=(110) pr_TPCdetail=(36)

where:

- **pr_errno** is the [EIO] error
- **pr_TPCerror** is the Guardian PROCESS_LAUNCH_ or PROCESS_CREATE_ error.

[ELOOP] Too many symbolic links were encountered in pathname resolution.

[EMFILE] The maximum number of files is open. The process attempted to open more than the maximum number of file descriptors allowed for the process. The process file segment (PFS) of the new process might be smaller than that of the calling process.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the pathname pointed to by the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENODEV] The system cannot find the device containing the fileset containing the process image file.

[ENOENT] One of the following conditions exists:

- One or more components of the new process image file's pathname do not exist.
- The *path* parameter points to an empty string.

[ENOEXEC] The new process image file has the appropriate access permissions, but it is neither in the correct binary executable format nor a valid executable text file.

[ENOMEM] Required resources are not available. Subsequent calls to the same function will not succeed for the same reason.

Possible causes of this error include insufficient primary memory (stack, globals, or heap) for the new process.

[ENOTDIR] A component of the path prefix of the new process image file is not a directory.

[ENOTOSS] The calling process is not an OSS process. A function in the **exec** set of functions cannot be called from the Guardian environment.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[ETXTBSY] The new process image file is currently open for writing by a process.

[EUNKNOWN]
Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

**RELATED INFORMATION**

Commands: **eld(1)**, **ld(1)**, **nld(1)**.

Functions: **alarm(3)**, **_exit(2)**, **execl(2)**, **execlp(2)**, **execv(2)**, **execve(2)**, **execvp(2)**, **execl(2)**, **fcntl(2)**, **fork(2)**, **getenv(3)**, **putenv(3)**, **semget(2)**, **sigaction(2)**, **system(3)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **times(3)**, **ulimit(3)**, **umask(2)**.

Miscellaneous: **environ(5)**.

**STANDARDS CONFORMANCE**

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- Guardian attributes are associated with the new OSS process. See **Guardian Attributes** under **DESCRIPTION**.
- The contents of the **st_atime** field following a failed function call in which the file was found should not be depended upon for further processing.
- The use of ***env[ ]** as a parameter in the call to **main( )** is an HP extension.

The following are HP extensions to the XPG4 Version 2 specification:

- Text files containing the **#!** *interpreter_name* **[***optional_string***]** header line can execute.
- The [EINVAL], [EIO], [ENODEV], [ENOTOSS], and [EUNKNOWN] error values are an HP extension.

## NAME

**execlp** - Executes a file using a filename, a set of argument strings, and ****environ**

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/yosskdll**

## SYNOPSIS

**#include <unistd.h>**

**extern char **environ;**

**int execlp(**
    **const char ****file***,**
    **const char ****arg***, ...);**

## PARAMETERS

****environ** Points to an array of character pointers to environment strings. The environment strings define the OSS environment for the new process. The **environ** array is terminated by a null pointer.

The ****environ** array of the new process is also passed as the *env***[ ]** array in the call to the **main( )** function of the new process. Refer to **Entering the New Process** later in this reference page.

*file* Identifies the new process image file. If this parameter

- Starts with a slash (/) character, then it contains the absolute pathname.
- Does not start with a slash but does contain a slash, then the pathname resolves relative to the current working directory.
- Contains no slash, the system searches the directories listed in the **PATH** environment variable for the file and prefixes the directory in which it is found.

*arg* Points to a null-terminated string containing an argument to be made visible to the main function of the new program. The first such argument should point to the null-terminated string containing the filename of the new process image. The last of these arguments must be a null pointer.

These strings constitute the argument list available to the new process image.

## DESCRIPTION

The **execlp( )** function is one of the **exec** set of functions. The **exec** set of functions replace the current process image with a new process image. The new image is constructed from a regular executable file, called a new process image file. The new process image file is formatted as an executable text or binary file in one of the formats recognized by the **exec** set of functions.

A successful **execlp( )** function call does not return, because the calling process image is overlaid by the new process image.

### Entering the New Process

When a program is executed as a result of a call to a function in the **exec** set of functions, it is entered as a function call as follows:

**int main(**
  **int** *argc***,**
  **char** **argv***[ ],**
  **char** **env***[ ]);**

Here, the *argc* parameter is the argument count, the *argv***[ ]** parameter is an array of character pointers to the arguments themselves, and *env***[ ]** is a pointer to a character array listing the environment variables. The *argv***[ ]** array is terminated by a null pointer. The null pointer is not counted in *argc*.

The arguments specified by a program with one of the **exec** set of functions are passed on to the new process image in the corresponding arguments to the **main( )** function.

The *env***[ ]** parameter for the main function is an HP extension and is not the preferred method of obtaining the environment variables for the new process. Use of the ****environ** array is the preferred method.

### Passing the Arguments and the Environment

The number of bytes available for the new process's combined argument and environment lists has a system-imposed limit. This limit, which includes the pointers and the null terminators on the strings, is available by calling the **sysconf(_SC_ARG_MAX)** function.

### Executing a Binary File

If the file specified in the function call is a binary executable file, the function loads the file directly.

### Executing a Text File

If the file specified in the function call is not a binary executable file, the function examines the file to determine whether it is an executable text file. The function checks for a header line in the following format:

**#!** *interpreter_name* [*optional_string*]

The **#!** notation identifies the file as an executable text file. The new process image filename is constructed from the process image filename in the *interpreter_name* string, treating it like the *path* parameter. The arguments passed to the new process are modified as follows:

- The *argv***[0]** parameter is set to the name of the interpreter.
- If the *optional_string* portion is present, *argv***[1]** is set to *optional_string*.
- The next element of *argv***[ ]** is set to the original value of *file*.
- The remaining elements of *argv***[ ]** are set to the original elements of *argv***[ ]**, starting with the second and subsequent values of the *arg* parameter.
- The first value of *arg* is discarded.

The **S_ISUID** and **S_ISGID** mode bits of an executable text file are honored. Those bits of the *interpreter_name* command interpreter are ignored.

### When the File Is Invalid

If the process image file is not a valid executable object, or if the text file does not contain the header line, the **execlp( )** function invokes the **/bin/sh** command interpreter as the new process image and pass the following arguments to it:

- *argv***[0]** is set to the string "sh".
- *argv***[1]** is set to the original value of the *file* parameter.
- The remaining elements of *argv***[ ]** are set to the second and subsequent values of the *arg* parameter.
- The first instance of *arg* is discarded.

**Open Files**

File descriptors open in the calling process image remain open in the new process image, except for those:

- Whose close-on-exec flag **FD_CLOEXEC** is set (see the **fcntl(2)** reference page)
- Opened using a Guardian function or procedure call

If the process file segment of the new process image is smaller than the process file segment of the calling process image and if the calling process image has a large number of file descriptors open, then the system might not be able to propagate all the open file descriptors to the new process image. When this situation occurs, the function call fails and **errno** is set to the value of [EMFILE].

For those file descriptors that remain open, all attributes of the open file descriptor, including file locks, remain unchanged. All directory streams are closed.

**Shared Memory**

Any attached shared memory segments are detached by a successful call to a function in the **exec** set of functions. Refer to the **shmat(2)** reference page for additional information about shared memory segment use.

**Semaphores**

Semaphore set IDs attached to a calling process are also attached to the new process. The new process also inherits the adjust-on-exit (**semadj**) values of the calling process.

Refer to the **semget(2)** reference page for additional information about semaphore use.

**Signals**

Signals set to:

- The default action (**SIG_DFL**) in the calling process image are set to the default action in the new process image.
- Be ignored (**SIG_IGN**) by the calling process image are set to be ignored by the new process image.
- Cause abnormal termination (**SIG_ABORT**) in the calling process image are set to that action in the new process image.
- Cause entry into the debugger (**SIG_DEBUG**) in the calling process image are set to that action in the new process image.
- Be caught by the calling process image are set to the default action in the new process image.

See the **signal(4)** reference page either online or in the *Open System Services System Calls Reference Manual*.

### User ID and Group ID

If the set-user-ID mode bit of the new process image file is set (see the **chmod(2)** reference page), the effective user ID of the new process image is set to the owner ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved-set-user ID and the saved-set-group ID) for use by the **setuid( )** function.

The **_POSIX_SAVED_IDS** flag is defined **TRUE**.

### OSS Attributes

The following OSS attributes of the calling process image are unchanged after successful completion of any of the **exec** set of functions:

- OSS process ID (PID)
- Parent process ID
- Process group ID
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- The time left until an alarm clock signal is posted (see the **alarm(3)** reference page)
- Current working directory
- Root directory
- File mode creation mask (see the **umask(2)** reference page)
- Process signal mask (see the **sigprocmask(2)** reference page)
- Pending signals (see the **sigpending(2)** reference page)
- The **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** fields of the **tms** structure
- File size limit (see the **ulimit(2)** reference page)

Upon successful completion of the function call, the **st_atime** field of the file is marked for update.

The POSIX.1 standard does not specify the effect on the **st_atime** field when the function call fails but does find the file. Likewise, the HP implementation does not guarantee the outcome. Under these circumstances, this field should not be used for further processing.

### Guardian Attributes

The newly created OSS process retains the following Guardian attributes of the process that calls one of the **exec** set of functions:

- Priority
- Processor on which the process executes

- Home terminal
- Job ID
- DEFINE mode switch
- Process access ID (PAID), unless the **S_ISUID** mode bit of the new image file is set
- Security group list
- Job ancestor or GMOM
- Unread system message index (PCBMCNT)

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Outstanding incoming and outgoing message limits

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Login, remote, and saveabend flags
- File creation mask

The Guardian attributes of the new process differ from those of the calling process in the following ways:

- Segments created or shared using Guardian procedure calls such as SEGMENT_ALLOCATE_ are not inherited.
- The program file is the file specified in the function call.
- The library file is specified in the program file.
- The new process does not inherit the caller's extended swap file (if any). For a G-series TNS process or an accelerated process, the extended data segment is managed by the Kernel Managed Storage Facility (KMSF).
- The process name for the new process is system-generated if the RUNNAMED option is set in the program file. Otherwise the process is unnamed.
- The size of the data segment of the new process is set in the program file.
- The remote login flag (PCBREMID) is set to off if the program file has had its **S_ISUID** mode bit set. Otherwise, the remote login flag is set the same as for the caller.
- The size of the extended data segment of the new process is set in the program file.
- The DEFINEs inherited by the new process depend on the setting of DEFINE mode in the caller. If DEFINE mode in the caller is ON, all the caller's DEFINEs are inherited. If DEFINE mode is OFF, no DEFINEs are inherited.
- The process identification number (PIN) of the new process is unrelated to that of the calling process. The PIN of the new process is unrestricted if both of the following are true:

  — The HIGHPIN flag is set in the program file and any user library file.

  — The PIN of the calling process was unrestricted.

  If the PIN of the new process is restricted, then the PIN is in the range 0 through 254.

- The creator access ID (CAID) is set to the process access ID (PAID) of the calling process.
- The PAID depends on whether the **S_ISUID** mode bit of the image file is set. If so, the PAID is based on the file owner ID. If not, the PAID is the same as for the caller. (The **S_ISUID** mode bit of the image file has no effect on the security group list.)
- The MOM field for the new process depends on whether the calling process is named. If so, the MOM field for the new process is set to the caller's ANCESTOR field. Otherwise, the MOM field for the new process is set to the caller's MOM field.
- System debugger selection for the new process is based on Inspect mode.
- Code breakpoints and memory breakpoints are not inherited.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Use From the Guardian Environment

If called from a Guardian process, the function call fails and **errno** is set to [ENOTOSS].

## RETURN VALUES

If the **execlp( )** function returns to the calling process image, an error has occurred; the return value is -1, and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the function sets **errno** to the corresponding value. For any of these error conditions, file descriptors marked close-on-exec are not closed, signals set to be caught are not set to the default action, and none of the following are changed:

- The value of the global variable **environ**
- The pointers contained within the global variable **environ**
- The elements pointed to by **environ** pointers
- The effective user ID of the current process
- The effective group ID of the current process

[E2BIG] The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit. The limit can be obtained by calling the **sysconf(_SC_ARG_MAX)** function.

[EACCES] One of the following conditions exists:

- Search permission is denied for the directory components of the pathname prefix to the process image file.
- The new process image file, any library file, or script file denies execution permission.
- The new process image file is not a regular file.

[EAGAIN] System resources such as disk space, process control block (PCB) space, MAPPOOL space, stack space, or PFS space are temporarily inadequate.

[EFAULT] An input address parameter is outside valid bounds limits.

[EINVAL] The new process image file is a binary executable file with invalid attributes.

[EIO] Some physical input or output error has occurred. Either a file cannot be opened because of an input or output error, or data has been lost during an input or output transfer. This value is used for errors on the object file of a loaded program or library, or during data transfer with a Guardian environment home terminal.

For systems running J06.07 and later J-series RVUs or H06.18 or later H-series RVUs, this error can also occur when the OSS file system is out of memory and one or more open files cannot be propagated from the parent process to the child process. In this case, if you are running a program from the shell with the shell reporting any errors, you might see an error like this:

/bin/-sh: /bin/ps: tdm_execve(): failed with unexpected error pr_errno=(4005) pr_TPCerror=(110) pr_TPCdetail=(36)

where:

- **pr_errno** is the [EIO] error
- **pr_TPCerror** is the Guardian PROCESS_LAUNCH_ or PROCESS_CREATE_ error.

[ELOOP] Too many symbolic links were encountered in pathname resolution.

[EMFILE] The maximum number of files is open. The process attempted to open more than the maximum number of file descriptors allowed for the process. The process file segment (PFS) of the new process might be smaller than that of the calling process.

[ENAMETOOLONG]
One of the following is too long:

- The filename pointed to by the *file* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the value specified for the *file* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENODEV] The system cannot find the device containing the fileset containing the process image file.

[ENOENT] The *file* parameter points to an empty string.

[ENOEXEC] The new process image file has the appropriate access permissions, but it is neither in the correct binary executable format nor a valid executable text file. The **/bin/sh** command interpreter could not be invoked as a substitute.

[ENOMEM] Required resources are not available. Subsequent calls to the same function will not succeed for the same reason.

Possible causes of this error include insufficient primary memory (stack, globals, or heap) for the new process.

[ENOTOSS] The calling process is not an OSS process. A function in the **exec** set of functions cannot be called from the Guardian environment.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[ETXTBSY] The new process image file is currently open for writing by a process.

[EUNKNOWN]
Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

**RELATED INFORMATION**

Commands: **nld(1)**.

Functions: **alarm(3)**, **_exit(2)**, **execl(2)**, **execle(2)**, **execv(2)**, **execve(2)**, **execvp(2)**, **fcntl(2)**, **fork(2)**, **getenv(3)**, **putenv(3)**, **semget(2)**, **sigaction(2)**, **system(3)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **times(3)**, **ulimit(3)**, **umask(2)**.

Miscellaneous: **environ(5)**.

**STANDARDS CONFORMANCE**

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- Guardian attributes are associated with the new OSS process. See **Guardian Attributes** under **DESCRIPTION**.
- [ENOENT] is returned in **errno** if the environment variable **PATH** is not defined when the **execlp( )** function is called.
- The contents of the **st_atime** field following a failed function call in which the file was found should not be depended upon for further processing.
- The use of ***env*[ ]** as a parameter in the call to **main( )** is an HP extension.

The following are HP extensions to the XPG4 Version 2 specification:

- Text files containing the **#!** *interpreter_name* **[***optional_string***]** header line can execute.
- The [EINVAL], [EIO], [ENODEV], [ENOTOSS], and [EUNKNOWN] error values are an HP extension.

## NAME

**execv -** Executes a file using a pathname, an *argv* array, and ****environ**

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

## SYNOPSIS

```
#include <unistd.h>

extern char **environ;

int execv(
        const char *path,
        char * const argv[ ]);
```

## PARAMETERS

****environ**
Points to an array of character pointers to environment strings. The environment strings define the OSS environment for the new process. The **environ** array is terminated by a null pointer.

The ****environ** array of the new process is also passed as the *env***[ ]** array in the call to the **main( )** function of the new process. Refer to **Entering the New Process** later in this reference page.

*path*
Points to a null-terminated string containing a pathname that identifies the new process image file. The pathname is absolute if it starts with a slash (/) character. Otherwise, the pathname is relative and is resolved by prefixing the current working directory.

If the final component of the *path* parameter names a symbolic link, the link is traversed and pathname resolution continues.

*argv***[ ]**
Specifies an array of character pointers to null-terminated strings containing arguments to be passed to the main function of the new program. *argv***[0]** should point to the null-terminated string containing the filename of the new process image. The last member of this array must be a null pointer.

These strings constitute the argument list available to the new process image.

## DESCRIPTION

The **execv( )** function is one of the **exec** set of functions. The **exec** set of functions replace the current process image with a new process image. The new image is constructed from a regular executable file, called a new process image file. The new process image file is formatted as an executable text or binary file in one of the formats recognized by the **exec** set of functions.

A successful **execv( )** function call does not return, because the calling process image is overlaid by the new process image.

### Entering the New Process

When a program is executed as a result of a call to a function in the **exec** set of functions, it is entered as a function call as follows:

```
int main(
        int argc,
        char *argv[ ],
        char *env[ ]);
```

Here, the *argc* parameter is the argument count, the *argv***[ ]** parameter is an array of character pointers to the arguments themselves, and *env***[ ]** is a pointer to a character array listing the environment variables. The *argv*[ ] array is terminated by a null pointer. The null pointer is not counted in *argc*.

The arguments specified by a program with one of the **exec** set of functions are passed on to the new process image in the corresponding arguments to the **main( )** function.

The *env*[ ] parameter for the main function is an HP extension and is not the preferred method of obtaining the environment variables for the new process. Use of the ****environ** array is the preferred method.

### Passing the Arguments and the Environment

The number of bytes available for the new process's combined argument and environment lists has a system-imposed limit. This limit, which includes the pointers and the null terminators on the strings, is available by calling the **sysconf(_SC_ARG_MAX)** function.

### Executing a Binary File

If the file specified in the function call is a binary executable file, the function loads the file directly.

### Executing a Text File

If the file specified in the function call is not a binary executable file, all forms of the function examine the file to determine whether it is an executable text file. The function checks for a header line in the following format:

**#!** *interpreter_name* [*optional_string*]

The **#!** notation identifies the file as an executable text file. The new process image filename is constructed from the process image filename in the *interpreter_name* string, treating it like the *path* parameter. The arguments passed to the new process are modified as follows:

- The *argv*[0] parameter is set to the name of the interpreter.
- If the *optional_string* portion is present, *argv*[1] is set to *optional_string*.
- The next element of *argv*[ ] is set to the original value of *path*).
- The remaining elements of *argv*[ ] are set to the original elements of *argv*[ ], starting with *argv*[1].
- The original *argv*[0] is discarded.

The **S_ISUID** and **S_ISGID** mode bits of an executable text file are honored. Those bits of the *interpreter_name* command interpreter are ignored.

### When the File Is Invalid

If the process image file is not a valid executable object, or if the text file does not contain the header line, the **execv( )** function call fails and **errno** is set to the value of [ENOEXEC].

### Open Files

File descriptors open in the calling process image remain open in the new process image, except for those:

- Whose close-on-exec flag **FD_CLOEXEC** is set (see the **fcntl(2)** reference page)
- Opened using a Guardian function or procedure call

If the process file segment of the new process image is smaller than the process file segment of the calling process image and if the calling process image has a large number of file descriptors open, then the system might not be able to propagate all the open file descriptors to the new process image. When this situation occurs, the function call fails and **errno** is set to the value of [EMFILE].

For those file descriptors that remain open, all attributes of the open file descriptor, including file locks, remain unchanged. All directory streams are closed.

**Shared Memory**

Any attached shared memory segments are detached by a successful call to a function in the **exec** set of functions. Refer to the **shmat(2)** reference page for additional information about shared memory segment use.

**Semaphores**

Semaphore set IDs attached to a calling process are also attached to the new process. The new process also inherits the adjust-on-exit (**semadj**) values of the calling process.

Refer to the **semget(2)** reference page for additional information about semaphore use.

**Signals**

Signals set to:

- The default action (**SIG_DFL**) in the calling process image are set to the default action in the new process image.
- Be ignored (**SIG_IGN**) by the calling process image are set to be ignored by the new process image.
- Cause abnormal termination (**SIG_ABORT**) in the calling process image are set to that action in the new process image.
- Cause entry into the debugger (**SIG_DEBUG**) in the calling process image are set to that action in the new process image.
- Be caught by the calling process image are set to the default action in the new process image.

See the **signal(4)** reference page either online or in the *Open System Services System Calls Reference Manual*.

**User ID and Group ID**

If the set-user-ID mode bit of the new process image file is set (see the **chmod(2)** reference page), the effective user ID of the new process image is set to the owner ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved-set-user ID and the saved-set-group ID) for use by the **setuid( )** function.

The **_POSIX_SAVED_IDS** flag is defined **TRUE**.

**OSS Attributes**

The following OSS attributes of the calling process image are unchanged after successful completion of any of the **exec** set of functions:

- OSS process ID (PID)
- Parent process ID
- Process group ID

- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- The time left until an alarm clock signal is posted (see the **alarm(3)** reference page)
- Current working directory
- Root directory
- File mode creation mask (see the **umask(2)** reference page)
- Process signal mask (see the **sigprocmask(2)** reference page)
- Pending signals (see the **sigpending(2)** reference page)
- The **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** fields of the **tms** structure
- File size limit (see the **ulimit(2)** reference page)

Upon successful completion of the function call, the **st_atime** field of the file is marked for update.

The POSIX.1 standard does not specify the effect on the **st_atime** field when the function call fails but does find the file. Likewise, the HP implementation does not guarantee the outcome. Under these circumstances, this field should not be used for further processing.

**Guardian Attributes**

The newly created OSS process retains the following Guardian attributes of the process that calls one of the **exec** set of functions:

- Priority
- Processor on which the process executes
- Home terminal
- Job ID
- DEFINE mode switch
- Process access ID (PAID), unless the **S_ISUID** mode bit of the new image file is set
- Security group list
- Job ancestor or GMOM
- Unread system message index (PCBMCNT)

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Outstanding incoming and outgoing message limits

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.

- Login, remote, and saveabend flags
- File creation mask

The Guardian attributes of the new process differ from those of the calling process in the following ways:

- Segments created or shared using Guardian procedure calls such as SEGMENT_ALLOCATE_ are not inherited.
- The program file is the file specified in the function call.
- The library file is specified in the program file.
- The new process does not inherit the caller's extended swap file (if any). For a G-series TNS process or an accelerated process, the extended data segment is managed by the Kernel Managed Storage Facility (KMSF).
- The process name for the new process is system-generated if the RUNNAMED option is set in the program file. Otherwise the process is unnamed.
- The size of the data segment of the new process is set in the program file.
- The remote login flag (PCBREMID) is set to off if the program file has had its **S_ISUID** mode bit set. Otherwise, the remote login flag is set the same as for the caller.
- The size of the extended data segment of the new process is set in the program file.
- The DEFINEs inherited by the new process depend on the setting of DEFINE mode in the caller. If DEFINE mode in the caller is ON, all the caller's DEFINEs are inherited. If DEFINE mode is OFF, no DEFINEs are inherited.
- The process identification number (PIN) of the new process is unrelated to that of the calling process. The PIN of the new process is unrestricted if both of the following are true:

  — The HIGHPIN flag is set in the program file and any user library file.

  — The PIN of the calling process was unrestricted.

  If the PIN of the new process is restricted, then the PIN is in the range 0 through 254.
- The creator access ID (CAID) is set to the process access ID (PAID) of the calling process.
- The PAID depends on whether the **S_ISUID** mode bit of the image file is set. If so, the PAID is based on the file owner ID. If not, the PAID is the same as for the caller. (The **S_ISUID** mode bit of the image file has no effect on the security group list.)
- The MOM field for the new process depends on whether the calling process is named. If so, the MOM field for the new process is set to the caller's ANCESTOR field. Otherwise, the MOM field for the new process is set to the caller's MOM field.
- System debugger selection for the new process is based on Inspect mode.
- Code breakpoints and memory breakpoints are not inherited.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Use From the Guardian Environment

If called from a Guardian process, the function call fails and **errno** is set to [ENOTOSS].

## RETURN VALUES

If the **execv( )** function returns to the calling process image, an error has occurred; the return value is -1, and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the function sets **errno** to the corresponding value. For any of these error conditions, file descriptors marked close-on-exec are not closed, signals set to be caught are not set to the default action, and none of the following are changed:

- The *argv*[ ] array of pointers
- The elements pointed to by this array
- The value of the global variable **environ**
- The pointers contained within the global variable **environ**
- The elements pointed to by **environ** pointers
- The effective user ID of the current process
- The effective group ID of the current process

[E2BIG] The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit. The limit can be obtained by calling the **sysconf(_SC_ARG_MAX)** function.

[EACCES] One of the following conditions exists:

- Search permission is denied for the directory components of the pathname prefix to the process image file.
- The new process image file, any library file, or script file denies execution permission.
- The new process image file is not a regular file.

[EAGAIN] System resources such as disk space, process control block (PCB) space, MAP-POOL space, stack space, or PFS space are temporarily inadequate.

[EFAULT] An input address parameter is outside valid bounds limits.

[EINVAL] The new process image file is a binary executable file with invalid attributes.

[EIO] Some physical input or output error has occurred. Either a file cannot be opened because of an input or output error, or data has been lost during an input or output transfer. This value is used for errors on the object file of a loaded program or library, or during data transfer with a Guardian environment home terminal.

For systems running J06.07 and later J-series RVUs or H06.18 or later H-series RVUs, this error can also occur when the OSS file system is out of memory and one or more open files cannot be propagated from the parent process to the child process. In this case, if you are running a program from the shell with the shell reporting any errors, you might see an error like this:

/bin/-sh: /bin/ps: tdm_execve(): failed with unexpected error pr_errno=(4005) pr_TPCerror=(110) pr_TPCdetail=(36)

where:

- **pr_errno** is the [EIO] error
- **pr_TPCerror** is the Guardian PROCESS_LAUNCH_ or PROCESS_CREATE_ error.

[ELOOP] Too many symbolic links were encountered in pathname resolution.

[EMFILE] The maximum number of files is open. The process attempted to open more than the maximum number of file descriptors allowed for the process. The process file segment (PFS) of the new process might be smaller than that of the calling process.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the pathname pointed to by the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENODEV] The system cannot find the device containing the fileset containing the process image file.

[ENOENT] One of the following conditions exists:

- One or more components of the new process image file's pathname do not exist.
- The *path* parameter points to an empty string.

[ENOEXEC] The new process image file has the appropriate access permissions, but it is neither in the correct binary executable format nor a valid executable text file.

[ENOMEM] Required resources are not available. Subsequent calls to the same function will not succeed for the same reason.

Possible causes of this error include insufficient primary memory (stack, globals, or heap) for the new process.

[ENOTDIR] A component of the path prefix of the new process image file is not a directory.

[ENOTOSS] The calling process is not an OSS process. A function in the **exec** set of functions cannot be called from the Guardian environment.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[ETXTBSY] The new process image file is currently open for writing by a process.

[EUNKNOWN]
Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

## RELATED INFORMATION

Commands: **eld(1)**, **ld(1)**, **nld(1)**.

Functions: **alarm(3)**, **_exit(2)**, **execl(2)**, **execle(2)**, **execlp(2)**, **execve(2)**, **execvp(2)**, **fcntl(2)**, **fork(2)**, **getenv(3)**, **putenv(3)**, **semget(2)**, **sigaction(2)**, **system(3)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **times(3)**, **ulimit(3)**, **umask(2)**.

Miscellaneous: **environ(5)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- Guardian attributes are associated with the new OSS process. See **Guardian Attributes** under **DESCRIPTION**.
- The contents of the **st_atime** field following a failed function call in which the file was found should not be depended upon for further processing.
- The use of ***env[ ]** as a parameter in the call to **main( )** is an HP extension.

The following are HP extensions to the XPG4 Version 2 specification:

- Text files containing the **#!** *interpreter_name* **[***optional_string***]** header line can execute.
- The [EINVAL], [EIO], [ENODEV], [ENOTOSS], and [EUNKNOWN] error values are an HP extension.

## NAME

**execve -** Executes a file using a pathname, an *argv* array, and an *envp* array

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

## SYNOPSIS

**#include <unistd.h>**

**extern char **environ;**

**int execve(**
**const char** ***path*,**
**char * const** *argv***[ ],**
**char * const** *envp***[ ]);**

## PARAMETERS

****environ** Points to an array of character pointers to environment strings. The environment strings define the OSS environment for the calling process. The **environ** array is terminated by a null pointer.

When the new process is created, the corresponding ****environ** array is not initialized for it with the content of the ****environ** array of the calling process. Instead, the *envp***[ ]** array is written into the ****environ** array of the new process when the **execve( )** function call is processed.

The ****environ** array of the new process is also passed as the *env***[ ]** array in the call to the **main( )** function of the new process. Refer to **Entering the New Process** later in this reference page.

*path* Points to a null-terminated string containing a pathname that identifies the new process image file. The pathname is absolute if it starts with a slash (/) character. Otherwise, the pathname is relative and is resolved by prefixing the current working directory.

If the final component of the *path* parameter names a symbolic link, the link is traversed and pathname resolution continues.

*argv***[ ]** Specifies an array of character pointers to null-terminated strings containing arguments to be passed to the main function of the new program. *argv***[0]** should point to the null-terminated string containing the filename of the new process image. The last member of this array must be a null pointer.

These strings constitute the argument list available to the new process image.

*envp***[ ]** Specifies an array of character pointers to null-terminated strings that describe the environment for the new process. The last member of this array must be a null pointer.

## DESCRIPTION

The **execve( )** function is one of the **exec** set of functions. The **exec** set of functions replace the current process image with a new process image. The new image is constructed from a regular executable file, called a new process image file. The new process image file is formatted as an executable text or binary file in one of the formats recognized by the **exec** set of functions.

A successful **execve( )** function call does not return, because the calling process image is overlaid by the new process image.

### Entering the New Process

When a program is executed as a result of a call to a function in the **exec** set of functions, it is entered as a function call as follows:

```
int main(
        int argc,
        char *argv[ ],
        char *env[ ]);
```

Here, the *argc* parameter is the argument count, the *argv***[ ]** parameter is an array of character pointers to the arguments themselves, and *env***[ ]** is a pointer to a character array listing the environment variables. The *argv***[ ]** array is terminated by a null pointer. The null pointer is not counted in *argc*.

The arguments specified by a program with one of the **exec** set of functions are passed on to the new process image in the corresponding arguments to the **main( )** function.

The *env***[ ]** parameter for the main function is an HP extension and is not the preferred method of obtaining the environment variables for the new process. Use of the ****environ** array is the preferred method.

### Passing the Arguments and the Environment

The number of bytes available for the new process's combined argument and environment lists has a system-imposed limit. This limit, which includes the pointers and the null terminators on the strings, is available by calling the **sysconf(_SC_ARG_MAX)** function.

### Executing a Binary File

If the file specified in the function call is a binary executable file, the function loads the file directly.

### Executing a Text File

If the file specified in the function call is not a binary executable file, the function examines the file to determine whether it is an executable text file. The function checks for a header line in the following format:

**#!** *interpreter_name* [*optional_string*]

The **#!** notation identifies the file as an executable text file. The new process image filename is constructed from the process image filename in the *interpreter_name* string, treating it like the *path* parameter. The arguments passed to the new process are modified as follows:

- The *argv***[0]** parameter is set to the name of the interpreter.
- If the *optional_string* portion is present, *argv***[1]** is set to *optional_string*.
- The next element of *argv***[ ]** is set to the original value of *path*.
- The remaining elements of *argv***[ ]** are set to the original elements of *argv***[ ]**, starting with *argv***[1]**.
- The original *argv***[0]** is discarded.

The **S_ISUID** and **S_ISGID** mode bits of an executable text file are honored. Those bits of the *interpreter_name* command interpreter are ignored.

### When the File Is Invalid

If the process image file is not a valid executable object, or if the text file does not contain the header line, the **execve( )** function call fails and **errno** is set to the value of [ENOEXEC].

### Open Files

File descriptors open in the calling process image remain open in the new process image, except for those:

- Whose close-on-exec flag **FD_CLOEXEC** is set (see the **fcntl(2)** reference page)
- Opened using a Guardian function or procedure call

If the process file segment of the new process image is smaller than the process file segment of the calling process image and if the calling process image has a large number of file descriptors open, then the system might not be able to propagate all the open file descriptors to the new process image. When this situation occurs, the function call fails and **errno** is set to the value of [EMFILE].

For those file descriptors that remain open, all attributes of the open file descriptor, including file locks, remain unchanged. All directory streams are closed.

### Shared Memory

Any attached shared memory segments are detached by a successful call to a function in the **exec** set of functions. Refer to the **shmat(2)** reference page for additional information about shared memory segment use.

### Semaphores

Semaphore set IDs attached to a calling process are also attached to the new process. The new process also inherits the adjust-on-exit (**semadj**) values of the calling process.

Refer to the **semget(2)** reference page for additional information about semaphore use.

### Signals

Signals set to:

- The default action (**SIG_DFL**) in the calling process image are set to the default action in the new process image.
- Be ignored (**SIG_IGN**) by the calling process image are set to be ignored by the new process image.
- Cause abnormal termination (**SIG_ABORT**) in the calling process image are set to that action in the new process image.
- Cause entry into the debugger (**SIG_DEBUG**) in the calling process image are set to that action in the new process image.
- Be caught by the calling process image are set to the default action in the new process image.

See the **signal(4)** reference page either online or in the *Open System Services System Calls Reference Manual*.

### User ID and Group ID

If the set-user-ID mode bit of the new process image file is set (see the **chmod(2)** reference page), the effective user ID of the new process image is set to the owner ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved-set-user ID and the saved-set-group ID) for use by the **setuid( )** function.

The **_POSIX_SAVED_IDS** flag is defined **TRUE**.

**OSS Attributes**

The following OSS attributes of the calling process image are unchanged after successful completion of any of the **exec** set of functions:

- OSS process ID (PID)
- Parent process ID
- Process group ID
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- The time left until an alarm clock signal is posted (see the **alarm(3)** reference page)
- Current working directory
- Root directory
- File mode creation mask (see the **umask(2)** reference page)
- Process signal mask (see the **sigprocmask(2)** reference page)
- Pending signals (see the **sigpending(2)** reference page)
- The **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** fields of the **tms** structure
- File size limit (see the **ulimit(2)** reference page)

Upon successful completion of the function call, the **st_atime** field of the file is marked for update.

The POSIX.1 standard does not specify the effect on the **st_atime** field when the function call fails but does find the file. Likewise, the HP implementation does not guarantee the outcome. Under these circumstances, this field should not be used for further processing.

**Guardian Attributes**

The newly created OSS process retains the following Guardian attributes of the process that calls one of the **exec** set of functions:

- Priority
- Processor on which the process executes
- Home terminal
- Job ID
- DEFINE mode switch
- Process access ID (PAID), unless the **S_ISUID** mode bit of the new image file is set

- Security group list
- Job ancestor or GMOM
- Unread system message index (PCBMCNT)

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Outstanding incoming and outgoing message limits

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Login, remote, and saveabend flags
- File creation mask

The Guardian attributes of the new process differ from those of the calling process in the following ways:

- Segments created or shared using Guardian procedure calls such as SEGMENT_ALLOCATE_ are not inherited.
- The program file is the file specified in the function call.
- The library file is specified in the program file.
- The new process does not inherit the caller's extended swap file (if any). For a G-series TNS process or an accelerated process, the extended data segment is managed by the Kernel Managed Storage Facility (KMSF).
- The process name for the new process is system-generated if the RUNNAMED option is set in the program file. Otherwise the process is unnamed.
- The size of the data segment of the new process is set in the program file.
- The remote login flag (PCBREMID) is set to off if the program file has had its **S_ISUID** mode bit set. Otherwise, the remote login flag is set the same as for the caller.
- The size of the extended data segment of the new process is set in the program file.
- The DEFINEs inherited by the new process depend on the setting of DEFINE mode in the caller. If DEFINE mode in the caller is ON, all the caller's DEFINEs are inherited. If DEFINE mode is OFF, no DEFINEs are inherited.
- The process identification number (PIN) of the new process is unrelated to that of the calling process. The PIN of the new process is unrestricted if both of the following are true:

  — The HIGHPIN flag is set in the program file and any user library file.

  — The PIN of the calling process was unrestricted.

  If the PIN of the new process is restricted, then the PIN is in the range 0 through 254.
- The creator access ID (CAID) is set to the process access ID (PAID) of the calling process.

- The PAID depends on whether the **S_ISUID** mode bit of the image file is set. If so, the PAID is based on the file owner ID. If not, the PAID is the same as for the caller. (The **S_ISUID** mode bit of the image file has no effect on the security group list.)
- The MOM field for the new process depends on whether the calling process is named. If so, the MOM field for the new process is set to the caller's ANCESTOR field. Otherwise, the MOM field for the new process is set to the caller's MOM field.
- System debugger selection for the new process is based on Inspect mode.
- Code breakpoints and memory breakpoints are not inherited.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Use From the Guardian Environment

If called from a Guardian process, the function call fails and **errno** is set to [ENOTOSS].

## RETURN VALUES

If the **execve( )** function returns to the calling process image, an error has occurred; the return value is -1, and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the function sets **errno** to the corresponding value. For any of these error conditions, file descriptors marked close-on-exec are not closed, signals set to be caught are not set to the default action, and none of the following are changed:

- The *argv***[ ]** array of pointers
- The *envp***[ ]** array of pointers
- The elements pointed to by these arrays
- The effective user ID of the current process
- The effective group ID of the current process

[E2BIG] The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit. The limit can be obtained by calling the **sysconf(_SC_ARG_MAX)** function.

[EACCES] One of the following conditions exists:

- Search permission is denied for the directory components of the pathname prefix to the process image file.
- The new process image file, any library file, or script file denies execution permission.
- The new process image file is not a regular file.

[EAGAIN] System resources such as disk space, process control block (PCB) space, MAPPOOL space, stack space, or PFS space are temporarily inadequate.

[EFAULT] An input address parameter is outside valid bounds limits.

[EINVAL] The new process image file is a binary executable file with invalid attributes.

[EIO] Some physical input or output error has occurred. Either a file cannot be opened because of an input or output error, or data has been lost during an input or output transfer. This value is used for errors on the object file of a loaded program or library, or during data transfer with a Guardian environment home terminal.

For systems running J06.07 and later J-series RVUs or H06.18 or later H-series RVUs, this error can also occur when the OSS file system is out of memory and one or more open files cannot be propagated from the parent process to the child process. In this case, if you are running a program from the shell with the shell reporting any errors, you might see an error like this:

/bin/-sh: /bin/ps: tdm_execve(): failed with unexpected error pr_errno=(4005) pr_TPCerror=(110) pr_TPCdetail=(36)

where:

- **pr_errno** is the [EIO] error
- **pr_TPCerror** is the Guardian PROCESS_LAUNCH_ or PROCESS_CREATE_ error.

[ELOOP] Too many symbolic links were encountered in pathname resolution.

[EMFILE] The maximum number of files is open. The process attempted to open more than the maximum number of file descriptors allowed for the process. The process file segment (PFS) of the new process might be smaller than that of the calling process.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the pathname pointed to by the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENODEV] The system cannot find the device containing the fileset containing the process image file.

[ENOENT] One of the following conditions exists:

- One or more components of the new process image file's pathname do not exist.
- The *path* parameter points to an empty string.

[ENOEXEC] The new process image file has the appropriate access permissions, but it is neither in the correct binary executable format nor a valid executable text file.

[ENOMEM] Required resources are not available. Subsequent calls to the same function will not succeed for the same reason.

Possible causes of this error include insufficient primary memory (stack, globals, or heap) for the new process.

[ENOTDIR] A component of the path prefix of the new process image file is not a directory.

[ENOTOSS] The calling process is not an OSS process. A function in the **exec** set of functions cannot be called from the Guardian environment.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[ETXTBSY] The new process image file is currently open for writing by a process.

[EUNKNOWN]
Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

**RELATED INFORMATION**

Commands: **eld(1)**, **ld(1)**, **nld(1)**.

Functions: **alarm(3)**, **_exit(2)**, **execl(2)**, **execle(2)**, **execlp(2)**, **execv(2)**, **execvp(2)**, **fcntl(2)**, **fork(2)**, **getenv(3)**, **putenv(3)**, **semget(2)**, **sigaction(2)**, **system(3)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **times(3)**, **ulimit(3)**, **umask(2)**.

Miscellaneous: **environ(5)**.

**STANDARDS CONFORMANCE**

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- Guardian attributes are associated with the new OSS process. See **Guardian Attributes** under **DESCRIPTION**.
- The contents of the **st_atime** field following a failed function call in which the file was found should not be depended upon for further processing.
- The use of ***env*[ ]** as a parameter in the call to **main( )** is an HP extension.

The following are HP extensions to the XPG4 Version 2 specification:

- Text files containing the **#!** *interpreter_name* **[***optional_string***]** header line can execute.
- The [EINVAL], [EIO], [ENODEV], [ENOTOSS], and [EUNKNOWN] error values are an HP extension.

## NAME

**execvp** - Executes a file using a filename, an *argv* array, and ****environ**

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

## SYNOPSIS

```
#include <unistd.h>

extern char **environ;

int execvp(
        const char *file,
        char * const argv[ ]);
```

## PARAMETERS

****environ**
: Points to an array of character pointers to environment strings. The environment strings define the OSS environment for the new process. The **environ** array is terminated by a null pointer.

  The ****environ** array of the new process is also passed as the *env***[ ]** array in the call to the **main( )** function of the new process. Refer to **Entering the New Process** later in this reference page.

*file*
: Identifies the new process image file. If this parameter

  - Starts with a slash (/) character, then it contains the absolute pathname.
  - Does not start with a slash but does contain a slash, then the pathname resolves relative to the current working directory.
  - Contains no slash, the system searches the directories listed in the **PATH** environment variable for the file and prefixes the directory in which it is found.

*argv***[ ]**
: Specifies an array of character pointers to null-terminated strings containing arguments to be passed to the main function of the new program. *argv***[0]** should point to the null-terminated string containing the filename of the new process image. The last member of this array must be a null pointer.

  These strings constitute the argument list available to the new process image.

## DESCRIPTION

The **execvp( )** function is one of the set of **exec** functions. The **exec** set of functions replace the current process image with a new process image. The new image is constructed from a regular executable file, called a new process image file. The new process image file is formatted as an executable text or binary file in one of the formats recognized by the **exec** set of functions.

A successful **execvp( )** function call does not return, because the calling process image is overlaid by the new process image.

### Entering the New Process

When a program is executed as a result of a call to a function in the **exec** set of functions, it is entered as a function call as follows:

**int main(**
  **int** *argc***,**
  **char** ∗*argv***[ ],**
  **char** ∗*env***[ ]);**

Here, the *argc* parameter is the argument count, the *argv***[ ]** parameter is an array of character pointers to the arguments themselves, and *env***[ ]** is a pointer to a character array listing the environment variables. The *argv***[ ]** array is terminated by a null pointer. The null pointer is not counted in *argc*.

The arguments specified by a program with one of the **exec** set of functions are passed on to the new process image in the corresponding arguments to the **main( )** function.

The *env***[ ]** parameter for the main function is an HP extension and is not the preferred method of obtaining the environment variables for the new process. Use of the ****environ** array is the preferred method.

### Passing the Arguments and the Environment

The number of bytes available for the new process's combined argument and environment lists has a system-imposed limit. This limit, which includes the pointers and the null terminators on the strings, is available by calling the **sysconf(_SC_ARG_MAX)** function.

### Executing a Binary File

If the file specified in the function call is a binary executable file, the function loads the file directly.

### Executing a Text File

If the file specified in the function call is not a binary executable file, the function examines the file to determine whether it is an executable text file. The function checks for a header line in the following format:

**#!** *interpreter_name* [*optional_string*]

The **#!** notation identifies the file as an executable text file. The new process image filename is constructed from the process image filename in the *interpreter_name* string, treating it like the *path* parameter. The arguments passed to the new process are modified as follows:

- The *argv***[0]** parameter is set to the name of the interpreter.
- If the *optional_string* portion is present, *argv***[1]** is set to *optional_string*.
- The next element of *argv***[ ]** is set to the original value of *file*.
- The remaining elements of *argv***[ ]** are set to the original elements of *argv***[ ]**, starting with *argv***[1]**.
- The original *argv***[0]** is discarded.

The **S_ISUID** and **S_ISGID** mode bits of an executable text file are honored. Those bits of the *interpreter_name* command interpreter are ignored.

### When the File Is Invalid

If the process image file is not a valid executable object, or if the text file does not contain the header line, the **execvp( )** function invokes the **/bin/sh** command interpreter as the new process image and pass the following arguments to it:

- *argv***[0]** is set to the string "sh".
- *argv***[1]** is set to the original value of the *file* parameter.
- The remaining elements of *argv***[ ]** are set to the original elements of *argv***[ ]**, starting with *argv***[1]**.
- The original value of *argv***[0]** is discarded.

**Open Files**

File descriptors open in the calling process image remain open in the new process image, except for those:

- Whose close-on-exec flag **FD_CLOEXEC** is set (see the **fcntl(2)** reference page)
- Opened using a Guardian function or procedure call

If the process file segment of the new process image is smaller than the process file segment of the calling process image and if the calling process image has a large number of file descriptors open, then the system might not be able to propagate all the open file descriptors to the new process image. When this situation occurs, the function call fails and **errno** is set to the value of [EMFILE].

For those file descriptors that remain open, all attributes of the open file descriptor, including file locks, remain unchanged. All directory streams are closed.

**Shared Memory**

Any attached shared memory segments are detached by a successful call to a function in the **exec** set of functions. Refer to the **shmat(2)** reference page for additional information about shared memory segment use.

**Semaphores**

Semaphore set IDs attached to a calling process are also attached to the new process. The new process also inherits the adjust-on-exit (**semadj**) values of the calling process.

Refer to the **semget(2)** reference page for additional information about semaphore use.

**Signals**

Signals set to:

- The default action (**SIG_DFL**) in the calling process image are set to the default action in the new process image.
- Be ignored (**SIG_IGN**) by the calling process image are set to be ignored by the new process image.
- Cause abnormal termination (**SIG_ABORT**) in the calling process image are set to that action in the new process image.
- Cause entry into the debugger (**SIG_DEBUG**) in the calling process image are set to that action in the new process image.
- Be caught by the calling process image are set to the default action in the new process image.

See the **signal(4)** reference page either online or in the *Open System Services System Calls Reference Manual*.

### User ID and Group ID

If the set-user-ID mode bit of the new process image file is set (see the **chmod(2)** reference page), the effective user ID of the new process image is set to the owner ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved-set-user ID and the saved-set-group ID) for use by the **setuid( )** function.

The **_POSIX_SAVED_IDS** flag is defined **TRUE**.

### OSS Attributes

The following OSS attributes of the calling process image are unchanged after successful completion of any of the **exec** set of functions:

- OSS process ID (PID)
- Parent process ID
- Process group ID
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- The time left until an alarm clock signal is posted (see the **alarm(3)** reference page)
- Current working directory
- Root directory
- File mode creation mask (see the **umask(2)** reference page)
- Process signal mask (see the **sigprocmask(2)** reference page)
- Pending signals (see the **sigpending(2)** reference page)
- The **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** fields of the **tms** structure
- File size limit (see the **ulimit(2)** reference page)

Upon successful completion of the function call, the **st_atime** field of the file is marked for update.

The POSIX.1 standard does not specify the effect on the **st_atime** field when the function call fails but does find the file. Likewise, the HP implementation does not guarantee the outcome. Under these circumstances, this field should not be used for further processing.

### Guardian Attributes

The newly created OSS process retains the following Guardian attributes of the process that calls one of the **exec** set of functions:

- Priority

- Processor on which the process executes
- Home terminal
- Job ID
- DEFINE mode switch
- Process access ID (PAID), unless the **S_ISUID** mode bit of the new image file is set
- Security group list
- Job ancestor or GMOM
- Unread system message index (PCBMCNT)

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Outstanding incoming and outgoing message limits

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Login, remote, and saveabend flags
- File creation mask

The Guardian attributes of the new process differ from those of the calling process in the following ways:

- Segments created or shared using Guardian procedure calls such as SEGMENT_ALLOCATE_ are not inherited.
- The program file is the file specified in the function call.
- The library file is specified in the program file.
- The new process does not inherit the caller's extended swap file (if any). For a G-series TNS process or an accelerated process, the extended data segment is managed by the Kernel Managed Storage Facility (KMSF).
- The process name for the new process is system-generated if the RUNNAMED option is set in the program file. Otherwise the process is unnamed.
- The size of the data segment of the new process is set in the program file.
- The remote login flag (PCBREMID) is set to off if the program file has had its **S_ISUID** mode bit set. Otherwise, the remote login flag is set the same as for the caller.
- The size of the extended data segment of the new process is set in the program file.
- The DEFINEs inherited by the new process depend on the setting of DEFINE mode in the caller. If DEFINE mode in the caller is ON, all the caller's DEFINEs are inherited. If DEFINE mode is OFF, no DEFINEs are inherited.
- The process identification number (PIN) of the new process is unrelated to that of the calling process. The PIN of the new process is unrestricted if both of the following are true:
  - The HIGHPIN flag is set in the program file and any user library file.

— The PIN of the calling process was unrestricted.

If the PIN of the new process is restricted, then the PIN is in the range 0 through 254.

- The creator access ID (CAID) is set to the process access ID (PAID) of the calling process.
- The PAID depends on whether the **S_ISUID** mode bit of the image file is set. If so, the PAID is based on the file owner ID. If not, the PAID is the same as for the caller. (The **S_ISUID** mode bit of the image file has no effect on the security group list.)
- The MOM field for the new process depends on whether the calling process is named. If so, the MOM field for the new process is set to the caller's ANCESTOR field. Otherwise, the MOM field for the new process is set to the caller's MOM field.
- System debugger selection for the new process is based on Inspect mode.
- Code breakpoints and memory breakpoints are not inherited.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Use From the Guardian Environment

If called from a Guardian process, the function call fails and **errno** is set to [ENOTOSS].

## RETURN VALUES

If the **execvp( )** function returns to the calling process image, an error has occurred; the return value is -1, and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the function sets **errno** to the corresponding value. For any of these error conditions, file descriptors marked close-on-exec are not closed, signals set to be caught are not set to the default action, and none of the following are changed:

- The *argv***[ ]** array of pointers
- The elements pointed to by this array
- The value of the global variable **environ**
- The pointers contained within the global variable **environ**
- The elements pointed to by **environ** pointers
- The effective user ID of the current process
- The effective group ID of the current process

[E2BIG] The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit. The limit can be obtained by calling the **sysconf(_SC_ARG_MAX)** function.

[EACCES] One of the following conditions exists:

- Search permission is denied for the directory components of the pathname prefix to the process image file.

- The new process image file, any library file, or script file denies execution permission.
- The new process image file is not a regular file.

[EAGAIN] System resources such as disk space, process control block (PCB) space, MAPPOOL space, stack space, or PFS space are temporarily inadequate.

[EFAULT] An input address parameter is outside valid bounds limits.

[EINVAL] The new process image file is a binary executable file with invalid attributes.

[EIO] Some physical input or output error has occurred. Either a file cannot be opened because of an input or output error, or data has been lost during an input or output transfer. This value is used for errors on the object file of a loaded program or library, or during data transfer with a Guardian environment home terminal.

For systems running J06.07 and later J-series RVUs or H06.18 or later H-series RVUs, this error can also occur when the OSS file system is out of memory and one or more open files cannot be propagated from the parent process to the child process. In this case, if you are running a program from the shell with the shell reporting any errors, you might see an error like this:

/bin/-sh: /bin/ps: tdm_execve(): failed with unexpected error pr_errno=(4005) pr_TPCerror=(110) pr_TPCdetail=(36)

where:

- **pr_errno** is the [EIO] error
- **pr_TPCerror** is the Guardian PROCESS_LAUNCH_ or PROCESS_CREATE_ error.

[ELOOP] Too many symbolic links were encountered in pathname resolution.

[EMFILE] The maximum number of files is open. The process attempted to open more than the maximum number of file descriptors allowed for the process. The process file segment (PFS) of the new process might be smaller than that of the calling process.

[ENAMETOOLONG]
One of the following is too long:

- The intermediate result of pathname resolution when a symbolic link is part of the value specified by the *file* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENODEV] The system cannot find the device containing the fileset containing the process image file.

[ENOENT] The *file* parameter points to an empty string.

[ENOEXEC] The new process image file has the appropriate access permissions, but it is neither in the correct binary executable format nor a valid executable text file. The **/bin/sh** command interpreter could not be invoked as a substitute.

[ENOMEM] Required resources are not available. Subsequent calls to the same function will not succeed for the same reason.

Possible causes of this error include insufficient primary memory (stack, globals, or heap) for the new process.

[ENOTDIR] A component of the path prefix of the new process image file is not a directory.

[ENOTOSS] The calling process is not an OSS process. A function in the **exec** set of functions cannot be called from the Guardian environment.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[ETXTBSY] The new process image file is currently open for writing by a process.

[EUNKNOWN]
Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

## RELATED INFORMATION

Commands: **eld(1)**, **ld(1)**, **nld(1)**.

Functions: **alarm(3)**, **_exit(2)**, **execl(2)**, **execle(2)**, **execlp(2)**, **execv(2)**, **execve(2)**, **fcntl(2)**, **fork(2)**, **getenv(3)**, **putenv(3)**, **semget(2)**, **sigaction(2)**, **system(3)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **times(3)**, **ulimit(3)**, **umask(2)**.

Miscellaneous: **environ(5)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- Guardian attributes are associated with the new OSS process. See **Guardian Attributes** under **DESCRIPTION**.
- [ENOENT] is returned in **errno** if the environment variable **PATH** is not defined when the **execvp( )** function is called.
- The contents of the **st_atime** field following a failed function call in which the file was found should not be depended upon for further processing.
- The use of ***env[ ]** as a parameter in the call to **main( )** is an HP extension.

The following are HP extensions to the XPG4 Version 2 specification:

- Text files containing the **#!** *interpreter_name* **[***optional_string***]** header line can execute.
- The [EINVAL], [EIO], [ENODEV], [ENOTOSS], and [EUNKNOWN] error values are an HP extension.

**NAME**

**_exit** - Terminates a process

**LIBRARY**

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series native Guardian processes: implicit libraries
H-series OSS processes: implicit libraries

**SYNOPSIS**

**#include <unistd.h>**

**void _exit(**
**int** *status***);**

**PARAMETERS**

*status* Indicates the status of the process.

**DESCRIPTION**

The **_exit( )** function terminates the calling process and causes the following to occur:

- All the file descriptors and directory streams that are open in the calling process are closed.
- All shared memory segments attached to the calling process are detached from it. The value of the **shm_nattch** field in the data structure associated with the shared memory identifier of each affected shared memory segment is decremented by 1. Refer to the **shmget(2)** reference page for more information.
- The **semadj** value established by the calling process for each semaphore is added to the **semval** value for that semaphore. Refer to the **semop(2)** reference page for more information.
- Terminating a process by exiting does not terminate its child processes. Instead, the parent process ID of all the calling process's child processes and zombie child processes is set to 1.
- If the parent process of the calling process is executing the **wait( )** or **waitpid( )** function, the parent is notified of the termination of the calling process and the low-order 8 bits (that is, bits 24 through 31) of the *status* parameter are made available to it.
- If the parent process is not executing the **wait( )** or **waitpid( )** function when the child process terminates, the parent can do so later and the child's status will be returned to it at that time. Meanwhile, the child process is transformed into a zombie process, and its parent process is sent a **SIGCHLD** signal to notify it of the termination of a child process.

  A zombie process is a process that occupies a slot in the process table but has no other space allocated to it either in user or kernel space. The process table slot that it occupies is partially overlaid with time-accounting information to be used by the **times( )** function. (See the **sys/proc.h** header file.)

  A process remains a zombie process until its parent process issues a call to the **wait( )** or **waitpid( )** function. At this time, the zombie process goes away and its process table entry is released.
- If the process is a controlling process, a **SIGHUP** signal is sent to each process in the foreground process group of the controlling terminal belonging to the calling process. The controlling terminal is dissociated from the session, allowing it to be acquired by a

new controlling process.

- If the exit of a process causes a process group to become orphaned, and if any member of the newly orphaned process group is stopped, then a **SIGHUP** signal followed by a **SIGCONT** signal is sent to each member of the orphaned process group.

- Locks set by the **fcntl( )** function are removed.

### Use From the Guardian Environment

The **_exit( )** function can be called from any Guardian process as well as from OSS processes. Guardian processes, however, have no OSS process ID (PID) and therefore have no wait considerations.

## NOTES

Open System Services currently does not support Common-Usage C.

## RETURN VALUES

The **_exit( )** function does not return.

## RELATED INFORMATION

Functions: **atexit(3)**, **close(2)**, **exit(3)**, **semop(2)**, **shmget(2)**, **sigaction(2)**, **times(3)**, **wait(2)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HPimplementation:

- Open System Services currently does not support Common-Usage C.

- Child processes of a terminated process are assigned a parent process ID of 1.

# Section 3. System Functions (f - i)

This section contains reference pages for Open System Services (OSS) system function calls with names that begin with **f** through **i**. These reference pages reside in the **cat2** directory and are sorted alphabetically by U.S. English conventions in this section.

## NAME

**fchmod** - Changes file-access permissions

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>      /* optional except for POSIX.1 */
#include <sys/stat.h>

int fchmod(
        int filedes,
        mode_t mode);
```

## PARAMETERS

*filedes* Specifies the file descriptor of an open file.

*mode* Specifies the bit pattern that determines the access permissions.

## DESCRIPTION

The **fchmod( )** function sets the access permissions of a file pointed to by the *filedes* parameter according to the bit pattern specified by the *mode* parameter. The **fchmod( )** function is like the **chmod( )** function except that it operates on a file descriptor instead of a pathname.

To change the file access permissions of a file or directory, the effective user ID of the process must match the super ID or the owner of the file, or its effective user ID or one of its group affiliations must qualify it for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group.

If **fchmod( )** is invoked by a process whose effective user ID does not equal the super ID or file owner, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000, respectively) are cleared.

If **fchmod( )** is invoked to set either or both of the set-user-ID and set-group-ID bits of the file mode (04000 and 02000 respectively), then any file privileges the file might have had are cleared.

See also "Accessing Files in Restricted-Access Filesets."

If the **S_ISVTX** bit is on for a directory, only processes with an effective user ID equal to the user ID of the file's owner or the directory's owner, or a process with appropriate privileges, can remove files from the directory.

A call to the **fchmod( )** function has no effect on the file descriptor for a file that is open at the time of the call. However, new openers of the file will be authorized by using the new access permissions that were specified in the call.

The *mode* parameter is constructed by logically ORing one or more of these symbols, which are defined in the **sys/stat.h** header file:

**S_ISUID** Sets the process's effective user ID to the user ID of the file's owner on execution.

**S_ISGID** Sets the process's effective group ID to the group ID of the file's group on execution.

**S_ISVTX** For a directory, permits modification to the directory only if the effective user ID of the process matches that of the file being accessed.

**S_IRWXU** Permits the file's owner to read, write, and execute the file (or to search the directory).

**S_IRUSR** Permits the file's owner to read the file.

**S_IWUSR** Permits the file's owner to write to the file.

**S_IXUSR** Permits the file's owner to execute the file (or to search the directory).

**S_IRWXG** Permits the file's group to read, write, and execute the file (or to search the directory).

**S_IRGRP** Permits the file's group to read the file.

**S_IWGRP** Permits the file's group to write to the file.

**S_IXGRP** Permits the file's group to execute the file (or to search the directory).

**S_IRWXO** Permits others to read, write, and execute the file (or to search the directory).

**S_IROTH** Permits others to read the file.

**S_IWOTH** Permits others to write to the file.

**S_IXOTH** Permits others to execute the file (or to search the directory).

**S_TRUST** Establishes that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers when the memory segment containing the buffers is not shared. This flag applies only to loadfiles for a TNS/E native process and can be set only by a user with appropriate privileges (the super ID).

**S_TRUSTSHARED**
Establishes that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers regardless of whether the memory segment containing the buffers is shared. This flag applies only to loadfiles for a TNS/E native process and can be set only by a user with appropriate privileges (the super ID).

The **S_ISUID** bit of the file is not changed by the call if the file specified by the *path* parameter resides on a node where the calling process is not logged in.

The **S_ISGID** bit of the file is cleared if all of these conditions are true:

- The named file is a regular file.
- The process does not have appropriate privileges.
- The file's group ID does not match the effective group ID of the process or one of the IDs of the process's group list.

Upon successful completion, the **fchmod( )** function marks the **st_ctime** field of the file for update.

### Access Control Lists (ACLs)

When you execute the **fchmod( )** function, you can change the effective permissions granted by optional entries in the ACL for a file. In particular, using the **fchmod( )** function to remove read, write, and execute permissions from a file owner, owning group, and all others works as expected, because the **fchmod( )** function affects the **class** entry in the ACL, limiting any access that can be granted to additional users or groups through optional ACL entries. To verify the effect, use **getacl** command on the file after the **fchmod( )** function completes and note that all optional (nondefault) ACL entries with nonzero permissions also have the comment **# effective:**---.

To set the permission bits of ACL entries, use the **acl( )** function instead of the **fchmod( )** function.

ACLs are not supported for symbolic links.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted. In a restricted-access fileset:

- The super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is not permitted to invoke this function on files that it does not own unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.
- Processes that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, if the executable file for the process does not have the PRIVSOARFOPEN file privilege, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000 respectively) of the file accessed by this function are cleared.
- Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use on Guardian Objects

Attempting to set the access permissions on a Guardian file (that is, a file in the **/G** file system) fails with **errno** set to [EINVAL].

### Use From the Guardian Environment

The **fchmod( )** function is one of a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **fchmod( )** function returns the value 0 (zero). Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occur, the **fchmod( )** function sets **errno** to the corresponding value:

[EBADF] The file descriptor *filedes* is not valid.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EINTR] A signal was caught during execution of the system call.

[EINVAL] One of these conditions exists:

- The value of the *mode* parameter is invalid.
- An attempt was made to set access permissions on a Guardian file (that is, a file in the **/G** file system).

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ENOENT] The program attempted an operation on a file that is open but that has been unlinked (and the attributes of the file are no longer alterable).

[ENOROOT] One of these conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote node, and communication with the remote name server has been lost.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] The effective user ID does not match the ID of the owner of the file, and the calling process does not have super ID privilege.

[EROFS] The file referred to by *filedes* resides on a read-only fileset.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Commands: **chmod(1)**, **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **chown(2)**, **fcntl(2)**, **fchown(2)**, **getgroups(2)**, **lchmod(2)**, **lchown(2)**, **mknod(2)**, **open(2)**, **open64(2)**, **read(2)**, **setfilepriv(2)**, **write(2)**.

Miscellaneous topics: **acl(5)**.

**STANDARDS CONFORMANCE**

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- To change file-access permissions, either the process must have the same effective user ID as the owner of the file or the process must have an effective user ID of the super ID.
- A call to the **fchmod( )** function has no effect on the file descriptor for a file that is open at the time of the call. However, new openers of the file are authenticated by using the new access permissions that were specified in the call.
- The errors [EINTR] and [EINVAL] can be detected.

HP extensions to the XPG4 Version 2 specification are:

- To change the file access permissions of a file or directory, the effective user ID of the process must match the super ID or the owner of the file, or the effective user ID or one of the group affiliations for the process must qualify the process for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group.
- The **errno** values [EIO], [EFSBAD], [ENOROOT], and [EOSSNOTRUNNING] can be returned.

## NAME

**fchown -** Changes the owner and group IDs of a file

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>

int fchown(
        int filedes,
        uid_t owner,
        gid_t group);
```

## PARAMETERS

*filedes* Specifies a valid file descriptor.

*owner* Specifies a numeric value representing the owner ID.

*group* Specifies a numeric value representing the group ID.

## DESCRIPTION

The **fchown( )** function changes the owner and group of a file pointed to by the *filedes* parameter. The **fchown( )** function is like the **chown( )** function except that it operates on a file descriptor instead of a pathname.

Only a process that has an effective user ID equal to the super ID or to the file owner, or that has an effective user ID or group affiliation qualifying for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group can use the **fchown( )** function to change the group of a file. However, processes that have an effective user ID equal to the file owner can only change the group of a file to a group to which they belong (their effective group or one of their supplementary groups).

If the **fchown( )** function is invoked by a process whose effective user ID does not equal the super ID, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000, respectively) are cleared.

See also "Accessing Files in Restricted-Access Filesets."

A process can change the value of the owner ID of an OSS file only if the effective user ID of the process gives the process appropriate privileges. A process can change the value of the file group ID if the effective user ID of the process matches the owner ID of the file or the process has appropriate privileges. A process without appropriate privileges can change the group ID of a file only to the value of its effective group ID or to a value in its group list. However, if the **fchown( )** function is successfully invoked on a file, the **S_ISGID** and **S_ISUID** bits of the **st_mode** field of the **stat** structure are cleared unless the user has appropriate privileges.

The **_POSIX_CHOWN_RESTRICTED** feature is enforced for any file in the OSS file system. Only processes with appropriate privileges can change owner IDs.

If the *owner* or *group* parameter is specified as -1 cast to the type of **uid_t** or **gid_t**, respectively, the corresponding ID of the file is unchanged. To change only one attribute, specify the other as -1.

Upon successful completion, the **fchown( )** function marks the **st_ctime** field of the file for update.

### Access Control Lists (ACLs)

A user can allow or deny specific individuals and groups access to a file by using an ACL on the file. When using the **fchown( )** function with ACLs, if the new owner and/or group of a file have optional ACL entries corresponding to **user:***uid***:***perm* or **group:***gid***:***perm* in the ACL for a file, those entries remain in the ACL but no longer have any effect because they are superseded by the **user::***perm* or **group::***perm* entries in the ACL.

ACLs are not supported for symbolic links.

For more information about ACLs, see the **acl(5)** reference page.

### Use on Guardian Objects

You can use the **fchown( )** function on Guardian disk files (that is, disk files in the **/G** file system). Attempts to change the ownership of other types of Guardian files fail and set **errno** to [EINVAL].

For Guardian disk files, Guardian security is used, and any user can pass file ownership to any other user. You must specify a value other than -1 for the *owner* parameter (that is, an owner ID must be specified). However, changing the owner ID also changes the group ID to the Guardian group ID of the new owner (that is, bits <16:23> of the new access ID). You cannot use the **fchown( )** function to set the group ID for a Guardian file except as a result of changing the owner ID.

The **_POSIX_CHOWN_RESTRICTED** feature is ignored for files in the Guardian file system (that is, for files in **/G**).

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted. In a restricted-access fileset:

- The super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is not permitted to invoke this function on files that it does not own unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.
- Processes that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, if the executable file for the process does not have the PRIVSOARFOPEN file privilege, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000 respectively) of the file accessed by this function are cleared.
- Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use From the Guardian Environment

The **fchown( )** function is one of a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **fchown( )** function returns the value 0 (zero). Otherwise, the value **-**1 is returned, the owner and group of the file remain unchanged, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occur, the **fchown( )** function sets **errno** to the corresponding value:

[EBADF] The file descriptor *filedes* is not valid.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EINTR] A signal was caught during execution of the system call.

[EINVAL] The *owner* or *group* parameter is out of range.

An attempt was made to change ownership of a Guardian file that is not a disk file.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ENOENT] The program attempted an operation on a file that is open but that has been unlinked (and the attributes of the file are no longer alterable).

[ENOROOT] One of these conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote node, and communication with the remote name server has been lost.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] The calling process does not have appropriate privileges.

[EROFS] The file referred to by *filedes* resides on a read-only fileset.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Commands: **chgrp(1)**, **chown(1)**, **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **chown(2)**, **fchmod(2)**, **lchmod(2)**, **lchown(2)**, **setfilepriv(2)**.

Miscellaneous topics: **acl(5)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- A process can change the value of the owner ID of a file only if the effective user ID of the process gives the process appropriate privileges.
- Upon successful completion, the set-user-ID attribute (the **S_ISUID** bit) and the set-group-ID attribute (the **S_ISGID** bit) of the file are always cleared.
- To change the file access permissions of a file or directory, the effective user ID of the process must match the super ID or the owner of the file, or the effective user ID or one of the group affiliations for the process must qualify the process for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group.
- The errors [EINTR], [EINVAL], and [EIO] can be detected.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EFSBAD], [EIO], [ENOROOT], and [EOSSNOTRUNNING] can be returned.

## NAME

**fcntl -** Controls open file descriptors

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <sys/types.h>    /* optional except for POSIX.1 */
#include <unistd.h>       /* optional except for POSIX.1 */
#include <fcntl.h>

int fcntl(
        int filedes,
        int request
        [, int argument1 |
         [, struct flock *argument2 |
          , struct flock64 *argument2]]);
```

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function, or the thread-aware **accept( )**, **dup2( )**, or **fcntl( )** function.

*request* Specifies the operation to be performed.

*argument1* Specifies a variable that depends on the value of the *request* parameter.

*argument2* Specifies a variable that depends on the value of the *request* parameter.

## DESCRIPTION

The **fcntl( )** function performs controlling operations on the open file specified by the *filedes* parameter.

Values for the *request* parameter are:

**F_DUPFD** Returns a new file descriptor as listed:

- Returns the lowest-numbered available file descriptor that is greater than or equal to the *argument1* parameter.
- References the same open file description as the original file descriptor.
- Returns the same file pointer as the original file (that is, both file descriptors share one file pointer if the object is a file).
- Returns the same access mode (read, write, or read/write).
- Returns the same file status flags (that is, both file descriptors share the same file status flags).

- Clears the close-on-exec flag (**FD_CLOEXEC** bit) associated with the new file descriptor so that the file remains open across calls to any function in the **exec**, **tdm_exec**, or **tdm_spawn** sets of functions.

The value **F_DUPFD** is invalid for an OSSTTY or Telserv terminal device. If this value is used in a call that specifies such a device for the *filedes* parameter, the call fails and **errno** is set to [EINVAL].

**F_GETFD** Gets the value of the file descriptor flags, defined in the **fcntl.h** header file, that are associated with the value of the *filedes* parameter. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file. The *argument1* parameter or *argument2* parameter is ignored.

The value **F_GETFD** is invalid for an OSSTTY or Telserv terminal device. If this value is used in a call that specifies such a device for the *filedes* parameter, the call fails and **errno** is set to [EINVAL].

**F_SETFD** Sets the value of the file descriptor flags, defined in the **fcntl.h** header file, that are associated with the *filedes* parameter to the value of the *argument1* parameter.

If the **FD_CLOEXEC** flag in the *argument1* parameter is 0 (zero), the file remains open across calls to any function in the **exec**, **tdm_exec**, and **tdm_spawn** sets of functions; otherwise, the file is closed on successful execution of the next function in an **exec**, **tdm_exec**, or **tdm_spawn** function set. When the **FD_CLOEXEC** flag is set, no other flag can be set in the call.

The value **F_SETFD** is invalid for an OSSTTY or Telserv terminal device. If this value is used in a call that specifies such a device for the *filedes* parameter, the call fails and **errno** is set to [EINVAL].

**F_GETFL** Gets the file status flags and file access modes, defined in the **fcntl.h** header file, for the file referred to by the *filedes* parameter.

The file access modes can be extracted by using the mask **O_ACCMODE** on the return value. File status flags and file access modes are associated with the file descriptor and do not affect other file descriptors that refer to the same file with different open file descriptors.

The *argument1* or *argument2* parameter is ignored.

The **O_APPEND**, **O_NONBLOCK**, and **O_SYNC** flags are not returned as set if they were ignored in a previous call using **F_SETFL**.

**F_SETFL** Sets the file status flags **O_APPEND**, **O_NONBLOCK**, and **O_SYNC** for the file to which the *filedes* parameter refers, from the corresponding bits in the *argument1* parameter. Some flags are ignored, depending on the file type, as listed:

Table 3–1. Ignored File Status Flags

| File type | Ignored file status flags |
|---|---|
| Directory | **O_APPEND**, **O_NONBLOCK**, **O_SYNC** |
| FIFO, pipe | **O_APPEND**, **O_SYNC** |
| Character special file | **O_APPEND**, **O_SYNC** |
| Regular file | **O_NONBLOCK** |
| Socket | **O_APPEND, O_SYNC** |

These file status flags are always accepted and ignored:

**O_ACCMODE**
**O_CREAT**
**O_EXCL**
**O_TRUNC**

The file access mode is not changed when **F_SETFL** is used.

**F_GETOWN** Gets the process ID or process group ID currently receiving the **SIGURG** signal for a socket. A process group ID is returned as a negative value. A positive value indicates the process ID.

The value **F_GETOWN** is invalid for these calls:

- Guardian use of OSS sockets is not supported. If this value is used in a call from the Guardian environment, the call fails, and **errno** is set to [ENOTOSS].
- If this value is used in a call that specifies anything other than a socket for the *filedes* parameter, the call fails, and **errno** is set to [EINVAL].

**F_SETOWN** Sets the process ID or process group ID to receive the **SIGURG** signal for a socket. A process group ID is specified by supplying it as a negative value in the *argument1* parameter; otherwise, the *argument1* parameter is interpreted as a process ID.

The value **F_SETOWN** is invalid for these calls:

- Guardian use of OSS sockets is not supported. If this value is used in a call from the Guardian environment, the call fails, and **errno** is set to [ENOTOSS].
- If this value is used in a call that specifies anything other than a socket for the *filedes* parameter, the call fails, and **errno** is set to [EINVAL].

These values listed for the *request* parameter are available for advisory record locking on regular files. Advisory record locking is supported only for regular files. If attempted on other files, the operation fails, and **errno** is set to [EINVAL].

**F_GETLK** Gets the first lock that blocks the lock description pointed to by the *argument2* parameter. The information retrieved overwrites the information passed to the **fcntl()** function in the **flock** structure. If no lock is found that would prevent this lock from being created, the structure is left unchanged except for the lock type, which is set to **F_UNLCK**.

**F_GETLK64** Similar to **F_GETLK**, except that it takes a pointer to a **flock64** structure instead of a pointer to a **flock** structure.

**F_SETLK** Sets or clears a file segment lock according to the lock description pointed to by the *argument2* parameter. **F_SETLK** is used to establish shared locks (**F_RDLCK**) or exclusive locks (**F_WRLCK**) and, additionally, to remove either type of lock (**F_UNLCK**). If a shared (read) or exclusive (write) lock cannot be set, the **fcntl( )** function returns immediately with the value -1.

**F_SETLK64** Similar to **F_SETLK**, except that it takes a pointer to a **flock64** structure instead of a pointer to a **flock** structure.

**F_SETLKW** Same as **F_SETLK** except that, if a shared or exclusive lock is blocked by other locks, the process waits until it is unblocked. If a signal is received while **fcntl( )** is waiting for a region, the function is interrupted, -1 is returned, and **errno** is set to [EINTR].

For regular files, if a thread-aware **fcntl( )** function needs to wait for **F_SETLKW** requests, the thread-aware **fcntl( )** blocks the thread that called the function (instead of blocking the entire process).

**F_SETLKW64**
Similar to **F_SETLKW**, except that it takes a pointer to a **flock64** structure instead of a pointer to a **flock** structure.

For regular files, if a thread-aware **fcntl( )** function needs to wait for **F_SETLKW64** requests, the thread-aware **fcntl( )** blocks the thread that called the function (instead of blocking the entire process).

The **O_NONBLOCK** file status flag affects only operations against file descriptors derived from the same **open( )** function.

When a shared lock is set on a segment of a file, other processes can set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock fails if the file descriptor is not opened with read access.

An exclusive lock prevents any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock fails if the file descriptor was not opened with write access.

The **flock** and **flock64** structures describe the type (**l_type** field), starting offset (**l_whence**), relative offset (**l_start**), size (**l_len**), and process ID (**l_pid**) of the segment of the file to be affected.

The value of **l_whence** is set to **SEEK_SET**, **SEEK_CUR**, or **SEEK_END** to indicate that the relative offset of **l_start** bytes is measured from the start of the file, from the current position, or from the end of the file, respectively. The value of **l_len** is the number of consecutive bytes to be locked. The **l_len** value can be negative (where the definition of type **off_t** permits negative values of **l_len**). The **l_pid** field is used only with **F_GETLK** or **F_GETLK64** to return the process ID of the process holding a blocking lock. After a successful **F_GETLK** or **F_GETLK64** request, the value of **l_whence** becomes **SEEK_SET**.

If **l_len** is positive, the area affected starts at **l_start** and ends at **l_start** + **l_len - 1**. If **l_len** is negative, the area affected starts at **l_start** + **l_len** and ends at **l_start - 1**. Lock lengths can be negative.

Locks can start and extend beyond the current end of a file, but they cannot be negative relative to the beginning of the file. If **l_len** is set to 0 (zero), a lock can be set to always extend to the largest possible value of the file offset for that file. If such a lock also has **l_start** set to 0 (zero) and **l_whence** is set to **SEEK_SET**, the whole file is locked.

Changing or unlocking a portion from the middle of a larger locked segment leaves a smaller segment at either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or when the process holding that file descriptor terminates. Locks are not inherited by a child process in a function like **fork( )**, **tdm_fork( )**, or **tdm_spawn( )**.

**NOTES**

To use the **fcntl( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_fcntlz(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the value returned by the **fcntl( )** function depends on the value of the *request* parameter, listed:

**F_DUPFD** Returns a new file descriptor.

**F_GETFD** Returns the value of the file descriptor flags. The return value is not negative.

**F_GETFL** Returns the value of file status flags and access modes. The return value is not negative.

**F_GETLK** Returns the value 0 (zero).

**F_GETLK64** Returns the value 0 (zero).

**F_GETOWN** Returns the process ID or process group ID of the socket receiving a **SIGURG** signal. A positive value is a process ID; a negative value is a process group ID.

**F_SETFD** Returns the value 0 (zero).

**F_SETFL** Returns the value 0 (zero).

**F_SETLK** Returns the value 0 (zero).

**F_SETLK64** Returns the value 0 (zero).

**F_SETLKW** Returns the value 0 (zero).

**F_SETLKW64**
Returns the value 0 (zero).

**F_SETOWN** Returns the value 0 (zero).

If the **fcntl( )** function fails, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **fcntl( )** function sets **errno** to the corresponding value:

[EAGAIN] The *request* parameter is **F_SETLK** or **F_SETLK64**, the type of lock (**l_type**) is shared (**F_RDLCK**) or exclusive (**F_WRLCK**), and a segment of a file to be locked is already exclusive-locked by another process.

The *request* parameter is is **F_SETLK** or **F_SETLK64**, the type of lock is exclusive, and some portion of a segment of a file to be locked is already shared-locked or exclusive-locked by another process.

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function is in progress on a regular file and a function that is process-blocking for regular files attempts to begin an I/O operation on the same open file.

If the **fcntl( )** function is thread-aware, the [EALREADY] value is not returned.

[EBADF] One of these conditions exists:

- The *request* parameter is **F_SETLK**, **F_SETLK64**, **F_SETLKW**,or **F_SETLKW64**, the type of lock is shared (**F_RDLCK**), and *filedes* is not a valid file descriptor open for reading.
- The type of lock is exclusive (**F_WRLCK**), and *filedes* is not a valid file descriptor open for writing.
- The *filedes* parameter is not a valid open file descriptor.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The peer socket forcibly closed the connection.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The *argument2* parameter is an invalid address.

[EINTR] The *request* parameter is **F_SETLKW** or **F_SETLKW64**, and the **fcntl( )** function was interrupted by a signal that was caught.

[EINVAL] One of these conditions exists:

- The *request* parameter is **F_DUPFD**, and the *argument1* parameter is negative or greater than or equal to the maximum number of open file descriptors permitted.
- The *request* parameter is **F_GETLK**, **F_GETLK64**, **F_SETLK**, **F_SETLK64**, **F_SETLKW**,or **F_SETLKW64**, and the data pointed to by *argument2* is invalid, or *filedes* refers to a file that does not support locking.
- The *request* parameter is **F_GETOWN**, and the *filedes* parameter does not specify a socket.
- The *request* parameter is **F_SETFD**, and a flag in addition to **FD_CLOEXEC** in the *argument1* parameter is set. When the *request* parameter is **F_SETFD** and **FD_CLOEXEC** is set, no other flag can be set.
- The *request* parameter is **F_SETFL**, and any file status flag other than **O_NONBLOCK**, **O_APPEND**, **O_CREAT**, **O_EXCL**, **O_SYNC**, or **O_TRUNC** is set. (Values set in the **O_ACCMODE** mask are ignored.)
- The *request* parameter is **F_SETOWN**, and the *filedes* parameter does not specify a socket.
- The call attempted to set an advisory record lock on a file that is not a regular file.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[EMFILE] The *request* parameter is **F_DUPFD** and the maximum number of open file descriptors permitted are currently open in the calling process, or no file descriptors greater than or equal to *argument1* are available.

[ENETDOWN]
The *request* parameter is **F_SETLK** or **F_SETLK64**, the *filedes* parameter specifies a file on a remote node, and communication with the remote node has been lost.

[ENOLCK] The *request* parameter is **F_SETLK**, **F_SETLK64**, **F_SETLKW**, or **F_SETLKW64**, and satisfying the lock or unlock request would cause the number of locked regions in the system to exceed a system-imposed limit.

[ENOTOSS] The *filedes* parameter specifies a socket, and the calling process is running in the Guardian environment. You cannot use the **fcntl( )** function on an OSS socket from the Guardian environment.

[EOVERFLOW]
The command argument is **F_GETLK**, **F_SETLK**, or **FSETLKW**, and the smallest offset (if *l_len* parameter is zero), or the highest offset (if the *l_len* parameter is nonzero), of any byte in the requested segment cannot be represented correctly in an object of type **off_t**.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and the backup process took over.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Functions: **creat(2)**, **creat64(2)**, **close(2)**, **dup(2)**, **dup2(2)**, **exec(2)**, **open(2)**, **open64(2)**, **read(2)**, **socket(2)**, **spt_fcntlz(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **write(2)**.

## STANDARDS CONFORMANCE

The **fcntl( )** function does not return the **errno** value [EDEADLK].

The **fcntl( )** function does not support the **O_ASYNC** flag.

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- Advisory record locking is supported only for regular files. If attempted on other files, the operation fails, and **errno** is set to [EINVAL].
- For record-locking operations, the **l_len** value can be negative (where the definition of **off_t** permits negative values of **l_len**). If **l_len** is negative, the area affected by the lock starts at **l_start** + **l_len** and ends at **l_start - 1**.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [ECONNRESET], [EFAULT], [EIO], [EISGUARDIAN], [ENETDOWN], [ENOTOSS], and [EWRONGID] can be returned.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**fork -** Creates a new process

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>

pid_t fork(void);
```

## DESCRIPTION

The **fork( )** function creates a new OSS process. The created process is referred to as the child and the caller as the parent. The child process executes the same program file as the parent and retains many other Guardian attributes as well as OSS attributes of the parent.

The **_POSIX_SAVED_IDS** flag is defined **TRUE**. The saved-set-user-ID and saved-set-group-ID fields of the parent process are therefore inherited by the child.

### Use From the Guardian Environment

If called from a Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

### OSS Attributes

The child process inherits the following OSS attributes from the parent process:

- Environment
- Close-on-exec flags
- Signal handling settings
- Saved-set-user-ID mode bit
- Saved-set-group-ID mode bit
- Process group ID
- Current directory
- Root directory
- File mode creation mask
- File size limit (see the **ulimit(2)** reference page)
- Attached shared memory segments
- Attached semaphore set IDs

The OSS attributes of the child process differ from those of the parent process in the following ways:

- The child process has a unique OSS process ID (PID) and does not match any active process group ID.

- The parent process ID of the child process matches the PID of the parent.
- The child process has its own copy of the parent process's file descriptors. However, each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent process.
- The child process does not inherit any file open created by a Guardian function or procedure call.
- The child process does not inherit file locks.
- The child process's **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** values are set to 0 (zero).
- Any pending alarms are cleared in the child process.
- Any signals pending for the parent process are not inherited by the child process.
- Any adjust-on-exit (**semadj**) values of the parent process are cleared in the child process.
- The child process shares directory streams with the parent. They share the same block of directory entries. When reading an entry, the buffer pointer is advanced by one entry. From the perspective of either process, an entry might be skipped.

  If both processes call the **readdir( )** function for a shared stream, the results are undefined. After such a call by both functions, another call to the **readdir( )** function by either process has undefined results.

**Guardian Attributes**

The child process inherits the following Guardian attributes from the parent process:

- Program file
- Any library files
- The size and contents of any instance data segments for native libraries
- Priority (the child process inherits the parent's current priority)
- Processor on which the process executes
- Home terminal
- For G-series TNS or accelerated processes, the size and contents of the data segment
- For G-series TNS or accelerated processes, the size and contents of the extended data segment

  The assignment of the data segment size is different from the assignment made when creating a new process with Guardian procedures.
- For native processes, the contents of the stack segment from the origin of the stack through the currently in-use location; the balance of the child process stack contains 0 (zero)
- For native processes, the size and contents of the globals-heap segment
- Job ID
- DEFINE mode

- Creator access ID (CAID)
- Process access ID (PAID)
- Security group list
- Job ancestor or GMOM
- Unread system message index count (PCBMCNT)

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Outstanding incoming and outgoing message limits

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Login, remote login, and saveabend flags
- File creation mask

The Guardian attributes of the child process differ from those of the parent process in the following ways:

- Segments created or shared using Guardian procedures such as SEGMENT_ALLOCATE_ are not inherited.
- The child process does not inherit the parent process extended swap file (if any). For a G-series TNS process or an accelerated process, the extended data segment is managed by the Kernel Managed Storage Facility (KMSF).
- The child's process name is system-generated if the RUNNAMED option is set in the program file. Otherwise the process is unnamed.
- The DEFINEs inheritance for the child depends on the parent's DEFINE mode.
- The process identification number (PIN) of the child process is unrelated to that of the parent process. The PIN of the child process is unrestricted if both of the following are true:
  - The HIGHPIN flag is set in the program file and any user library file.
  - The PIN of the parent process was unrestricted.

  If the PIN of the child process is restricted, then the PIN is in the range 0 through 254.
- The MOM field for the child process is set to 0 (zero).
- System debugger selection for the child process is based on Inspect mode and OSS read access rights on the program file.
- Code breakpoints and memory breakpoints are not inherited.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Sharing Guardian Files

After a successful call to the **fork( )** function, the initial position within an EDIT file (file code 101) in the Guardian file system (a file in **/G**) that was opened by a call to the OSS **open( )** function is the same for both the parent and child processes. However, the position is not shared; that is, changing the position used by one process does not change the position used by the other process.

### Floating-Point Data

If the parent process uses IEEE floating-point data, the child process inherits all of the floating-point register contents of the parent process and any computation started before the **fork( )** function call completes in the child process. The contents of the status and control register are also inherited.

### Use From a Threaded Application

The thread-aware version of the **fork( )** function call creates a new process from the current thread.

## NOTES

To use the **fork( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_fork(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the **fork( )** function returns the value 0 (zero) to the child process and returns the PID of the child process to the parent process. If the **fork( )** function fails, the value -1 is returned to the parent process, no child process is created, and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **fork( )** function sets **errno** to the corresponding value:

[EACCES] Open for execute access on the code file or any library file was denied.

[EAGAIN] System resources such as disk space, process control block (PCB) space, MAP-POOL space, stack space, or PFS space are temporarily inadequate.

[EIO] Some physical input or output error has occurred. Either a file cannot be opened because of an input or output error or data has been lost during an input or output transfer. This value is used only for errors on the object file of a loaded program or library, or during data transfer with a Guardian environment home terminal.

[ENOMEM] Required resources are not available. Subsequent calls to the same function will not succeed for the same reason.

Possible causes of this error include insufficient primary memory (stack, globals, or heap) for the new process.

[ENOTOSS] The calling process is not an OSS process. The **fork( )** function cannot be called from the Guardian environment.

[EUNKNOWN]

Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

**RELATED INFORMATION**

Functions: **exec(2)**, **_exit(2)**, **exit(3)**, **raise(3)**, **semop(2)**, **shmat(2)**, **sigaction(2)**, **spt_fork(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **times(3)**, **ulimit(3)**, **umask(2)**, **wait(2)**.

**STANDARDS CONFORMANCE**

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- The child process shares directory streams with the parent. They share the same block of directory entries, but each stream can be used by only one of the processes.
- Guardian attributes are associated with the new OSS process. See **Guardian Attributes** under **DESCRIPTION**.

The following are HP extensions to the XPG4 Version 2 specification:

- The [EFAULT], [ENOTOSS], and [EUNKNOWN] error values are HP extensions.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

This function does not conform to the async-signal safe requirement of the POSIX.1 standard.

## NAME

**fstat -** Provides information about an open file

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <sys/types.h>    /* optional except for POSIX.1 */
#include <sys/stat.h>

int fstat(
        int filedes,
        struct stat *buffer);
```

## PARAMETERS

*filedes*
Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**,**dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function, or the thread-aware **accept( )**, **dup2( )**, or **fcntl( )** function.

*buffer*
Points to a **stat** structure, into which information is placed about the file. The **stat** structure is described in the **sys/stat.h** header file.

## DESCRIPTION

The **fstat( )** function obtains information about the open file associated with the *filedes* parameter.

The file information is written to the area specified by the *buffer* parameter, which is a pointer to a **stat** structure. For J06.11 and later J-series RVUs and H06.22 and later H-series RVUs, the **stat** structure uses this definition from the **sys/stat.h** header file:

```
struct stat {
        dev_t     st_dev;
        ino_t     st_ino;
        mode_t    st_mode;
        nlink_t   st_nlink;
        unsigned int      st_acl:1;
        unsigned int      __filler_1:7;
        unsigned int      st_fileprivs:8; /* File privileges */
        uid_t     st_uid;
        gid_t     st_gid;
#if _FILE_OFFSET_BITS != 64 || _TANDEM_ARCH_ == 0
        mode_t    st_basemode; /* Permissions with original group perms */
#endif
        dev_t     st_rdev;
        off_t     st_size;
        time_t    st_atime;
        time_t    st_mtime;
        time_t    st_ctime;
#if _FILE_OFFSET_BITS == 64 && _TANDEM_ARCH_ != 0
```

```
 mode_t st_basemode; /* Permissions with original group perms */
#endif
        int64_t   st_reserved[3];
};
```

For J06.10 and earlier J-series RVUs and H06.21 and earlier H-series RVUs, the **stat** structure uses this definition from the **sys/stat.h** header file:

```
struct  stat {
        dev_t     st_dev;
        ino_t     st_ino;
        mode_t    st_mode;
        nlink_t   st_nlink;
        unsigned int        st_acl:1;
        unsigned int        __filler_1:15;
        uid_t     st_uid;
        gid_t     st_gid;
#if _FILE_OFFSET_BITS != 64 || _TANDEM_ARCH_ == 0
        mode_t    st_basemode; /* Permissions with original group perms */
#endif
        dev_t     st_rdev;
        off_t     st_size;
        time_t    st_atime;
        time_t    st_mtime;
        time_t    st_ctime;
#if _FILE_OFFSET_BITS == 64 && _TANDEM_ARCH_ != 0
        mode_t    st_basemode; /* Permissions with original group perms */
#endif
        int64_t   st_reserved[3];
};
```

The **fstat( )** function updates any time-related fields associated with the file before writing into the **stat** structure, unless it is a read-only fileset. Time-related fields are not updated for read-only OSS filesets.

The fields in the **stat** structure have these meanings and content:

**st_dev** OSS device identifier for a fileset.

Values for local OSS objects are listed next. Values for local Guardian objects are described in **Use on Guardian Objects**, and values for remote Guardian or OSS objects are described in **Use on Remote Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | ID of device containing directory entry |
| Directory | ID of device containing directory |
| Pipe or FIFO | ID of special fileset for pipes |
| **AF_INET** or **AF_INET6** socket | ID of special fileset for sockets |
| **AF_UNIX** socket | ID of device containing the fileset in which the socket file was created |

| | |
|---|---|
| **/dev/null** | ID of device containing directory entry |
| **/dev/tty** | ID of device containing directory entry |

**st_ino** File serial number (inode number). The file serial number and OSS device identifier uniquely identify a regular OSS file within an OSS fileset.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | File serial number (unique) |
| Directory | File serial number (unique) |
| Pipe or FIFO | File serial number (unique) |
| **AF_INET** or **AF_INET6** socket | File serial number (not unique within the HP NonStop node) |
| **AF_UNIX** socket | File serial number of the socket file (unique) |
| **/dev/null** | File serial number (unique) |
| **/dev/tty** | File serial number (unique) |

The **st_ino** value for all node entries in **/E** (including the entry for the logical link from the local node name to the root fileset on the local node) is the value for the root fileset on the corresponding node. If normal conventions are followed, this value is always 0 (zero), so entries in **/E** appear to be nonunique. Values for objects on remote nodes are unique only among the values for objects within the same fileset on that node.

**st_mode** File mode. These bits are ORed into the **st_mode** field:

**S_IFMT** File type. This field can contain one of these values:

**S_IFCHR** Character special file.

**S_IFDIR** Directory.

**S_IFIFO** Pipe or FIFO.

**S_IFREG** Regular file.

**S_IFSOCK** Socket.

For an **AF_INET** or **AF_INET6** socket, the user default permissions are returned for the permission bits. The access flags are set to read and write.

For an **AF_UNIX** socket, the user permissions from the inode for the socket are returned for the permission bits. The access flags are also returned from the inode.

**S_IRWXG** Permissions for the owning group, or if the **st_acl** flag is set, permissions for the the **class** ACL entry.

**S_IRWXO** Other class

**S_IRWXU** Owner class

**S_ISGID** Set group ID on execution

**S_ISUID** Set user ID on execution

**S_ISVTX** Sticky bit; used only for directories (not ORed for files in **/G**, the Guardian file system)

**S_TRUST** Indicates that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers when the memory segment containing the buffers is not shared. This flag applies only to loadfiles for a process, and only a user with appropriate privileges (the super ID) can set it.

**S_TRUSTSHARED**
Indicates that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers regardless of whether the memory segment containing the buffers is shared. This flag applies only to loadfiles for a process, and only a user with appropriate privileges (the super ID) can set it.

Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

**st_nlink** Number of links.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Number of links to the file |
| Directory | Number of links to the directory |
| FIFO | Number of links to the file |
| Pipe | -1 |
| **AF_INET** or **AF_INET6** socket | 0 (zero) |
| **AF_UNIX** socket | Number of links to the socket file |
| **/dev/null** | Number of links to the file |
| **/dev/tty** | Number of links to the file |

**st_acl** If set to 1, indicates that the file has optional access control list (ACL) entries. For compatibility with HP-UX, the member name **st_aclv** is provided as alias for **st_acl**. For more information about ACLs, see the **acl(5)** reference page.

**st_fileprivs** File privileges. For information about file privileges see the **setfilepriv(2)** reference page.

**st_uid** User ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | User ID of the file owner |
| Directory | User ID of the file owner |
| Pipe or FIFO | User ID of the file owner |
| **AF_INET** or **AF_INET6** socket | User ID of the calling process |
| **AF_UNIX** socket | User ID of the creator of the socket file |
| **/dev/null** | User ID of the super ID |
| **/dev/tty** | User ID of the super ID |

**st_gid** Group ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Group ID of the file group |
| Directory | Group ID of the file group |
| Pipe or FIFO | Group ID of the file group |
| **AF_INET** or **AF_INET6** socket | Group ID of the calling process |
| **AF_UNIX** socket | Group ID of the creator of the socket file |
| **/dev/null** | Group ID of the super ID |
| **/dev/tty** | Group ID of the super ID |

**st_basemode** If the **st_acl** flag is set, contains the permissions for the file owner, owning group, and others. If the **st_acl** flag is not set, **st_basemode** is 0 (zero).

**st_rdev** Remote device ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Undefined |
| Directory | Undefined |
| Pipe or FIFO | Undefined |
| **AF_INET** or **AF_INET6** socket | 0 (zero) |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | Undefined |
| **/dev/tty** | ID of the device |

**st_size** File size.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Size of the file in bytes |
| Directory | 4096 |
| Pipe or FIFO | 0 (zero) |
| **AF_INET** or **AF_INET6** socket | 0 (zero) |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | 0 (zero) |
| **/dev/tty** | 0 (zero) |

**st_atime** Access time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last access |
| Directory | Time of the last access |
| Pipe or FIFO | Time of the last access |
| **AF_INET** or **AF_INET6** socket | Value maintained in the socket data structure |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_mtime** Modification time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last data modification |
| Directory | Time of the last modification |
| Pipe or FIFO | Time of the last data modification |
| **AF_INET** or **AF_INET6** socket | Value maintained in the socket data structure |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_ctime** Status change time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Time of the last file status change |
| Directory | Time of the last file status change |
| Pipe or FIFO | Time of the last file status change |
| **AF_INET** or **AF_INET6** socket | Value maintained in the socket data structure |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**Use on Guardian Objects**

The **st_dev** and **st_ino** fields of the **stat** structure do not uniquely identify Guardian files (files in **/G**).

The **st_dev** field is unique for **/G**, for each disk volume, and for each Telserv process (or other process of subdevice type 30), because each of these is a separate fileset.

The **S_ISGUARDIANOBJECT** macro can indicate whether an object is a Guardian object when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is a Guardian object and **FALSE** otherwise.

The **st_ino** field is a nonunique encoding of the Guardian filename.

The **st_rdev** field contains a unique minor device number for each entry in **/G/ztnt/**, representing each Telserv process subdevice.

The **st_size** field of an EDIT file (file code 101) is the actual (physical) end of file, not the number of bytes in the file. For directories, **st_size** is set to 4096.

When an OSS function is called for a Guardian EDIT file, the **st_mtime** field is set to the last modification time. The **st_atime** field indicates the last time the file was opened, and the **st_ctime** field is set equal to **st_mtime**. No other time-related fields are updated by OSS function calls.

The **st_ctime** and **st_atime** fields for Guardian regular disk files (except for EDIT files) are updated by OSS function calls, not by Guardian procedure calls.

The time fields for **/G**, **/G/***vol*, and **/G/***vol***/***subvol* always contain the current time.

The mapping between Guardian files and their corresponding file types described in the **st_mode** field is listed next:

| Example in /G | Guardian File Type | st_mode File Type | Permissions |
|---|---|---|---|
| **/G** | N/A | Directory | r-xr-xr-x |
| *vol* | Disk volume | Directory | rwxrwxrwx |
| *vol/subvol* | Subvolume | Directory | rwxrwxrwx |
| *vol/subvol/fileid* | Disk file | Regular file | See following text |
| *vol*/**#123** | Temporary disk file | Regular file | See following text |
| **ztnt** | Subtype 30 process | Directory | --x--x--x |
| **ztnt/#pty0001** | Subtype 30 process with qualifier | Character special | rw-rw-rw- |
| **vol1/zyq00001** | Subvolume | Directory | --------- |

A Guardian file classified as a directory is always owned by the super ID.

Guardian permissions are mapped as follows:

- Guardian network or any user permission is mapped to OSS other permission.
- Guardian community or group user permission is mapped to OSS group permission.
- Guardian user or owner permission is mapped to OSS owner permission.
- Guardian super ID permission is OSS super ID permission.
- Guardian read permission is mapped to OSS read permission.
- Guardian write permission is mapped to OSS write permission.
- Guardian execute permission is mapped to OSS execute permission.
- Guardian purge permission is ignored.

Users are not allowed read access to Guardian processes.

OSS file permissions are divided into three groups (owner, group, and other) of three permission bits each (read, write, and execute). The OSS permission bits do not distinguish between remote and local users as Guardian security does; local and remote users are treated alike.

### Use on Remote Objects

The content of the **st_dev** field of the **stat** structure is unique for each node in **/E** because each node is a separate fileset. Values for directories within **/E** are the same as values for objects on the local HP NonStop node.

The **S_ISEXPANDOBJECT** macro can indicate whether an object in the **/E** directory is on a remote HP NonStop server node when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is on a remote HP NonStop node and **FALSE** otherwise.

### Use From a Threaded Application

This function serializes file operations on an open file. If a thread calls **fstat( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

## NOTES

For J06.08 and earlier J-series RVUs, H06.19 and earlier H-series RVUs, or G-series RVUs, the OSS Network File System (NFS) cannot access OSS objects that have OSS ACLs that contain optional ACL entries.

For J06.09 and later J-series RVUs and H06.20 and later H-series RVUs, access by the OSS Network File System (NFS) to OSS objects that have OSS ACLs that contain optional ACL entries can be allowed, depending upon the NFSPERMMAP attribute value for the fileset that contains the object. For more information about NFS and ACLs, see the **acl(5)** reference page.

To use the **fstat( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_fstatz(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **fstat( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not a valid file descriptor.

[EFAULT] The *buffer* parameter points to a location outside of the allocated address space of the process.

[EFSBAD] The program attempted an operation involving a fileset with a corrupted fileset catalog.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOROOT] The program attempted an operation while the root fileset was unavailable.

[ENXIO] An invalid device or address was specified during an input or output operation on a special file. One of these events occurred:

- A device was specified that does not exist, or a request was made beyond the limits of the device.
- The fileset containing the requestor's current working directory or root directory is not mounted. This error can occur after failure and restart of an OSS name server process until the fileset has been repaired and remounted.
- The file size (in bytes) or the file inode number (serial number) cannot be represented correctly in the structure pointed to by the *buffer* parameter.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation on an input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

**RELATED INFORMATION**

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **chown(2)**, **fstat64(2)**, **link(2)**, **mknod(2)**, **open(2)**, **open64(2)**, **pipe(2)**, **setfilepriv(2)**, **spt_fstatz(2)**, **utime(2)**.

Miscellaneous Topics: **acl(5)**.

**STANDARDS CONFORMANCE**

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- For files other than regular disk files, the **st_size** field of the **stat** structure is set to 0 (zero). For directories, **st_size** is set to 4096.

- The **S_IRWXU**, **S_IRWXG**, **S_IRWXO**, **S_IFMT**, **S_ISVTX**, **S_ISGID**, and **S_ISUID** bits are ORed into the **st_mode** field of the **stat** structure.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EFAULT], [EFSBAD], [EIO], [EISGUARDIAN], [ENETDOWN], [ENOROOT], [ENXIO], and [EWRONGID] can be returned by the **fstat( )** function.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**fstat64 -** Provides information about an open file

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <sys/types.h>** /* optional except for POSIX.1 */
**#include <sys/stat.h>**

**int fstat64(**
 **int** *filedes***,**
 **struct stat64 ****buffer***);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**,**dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function, or the thread-aware **accept( )**, **dup2( )**, or **fcntl( )** function.

*buffer* Points to a **stat64** structure, into which information is placed about the file. The **stat64** structure is described in the **sys/stat.h** header file.

## DESCRIPTION

The **fstat64( )** function is similar to the **fstat( )** function except that, in addition to supporting smaller files, the **fstat64( )** function supports OSS files larger than approximately 2 gigabytes.

An application can explicitly call this function you compile the application using the **#define _LARGEFILE64_SOURCE 1** feature test macro or an equivalent compiler command option.

An application call to **fstat( )** is automatically mapped to this function when you compile the application using the **#define _FILE_OFFSET_BITS 64** feature test macro or an equivalent compiler command option.

The **fstat64( )** function obtains information about the open file associated with the *filedes* parameter.

The file information is written to the area specified by the *buffer* parameter, which is a pointer to a **stat64** structure which is a pointer to a **stat64** structure. For J06.11 and later J-series RVUs and H06.22 and later H-series RVUs, the **stat64** structure uses this definition from the **sys/stat.h** header file:

```
struct  stat64 {
        dev_t      st_dev;
        ino64_t    st_ino;
        mode_t     st_mode;
        nlink_t    st_nlink;
        unsigned int       __filler_1:7;
        unsigned int       st_fileprivs:8; /* File privileges */
        uid_t      st_uid;
        gid_t      st_gid;
        dev_t      st_rdev;
        off64_t    st_size;
        time_t     st_atime;
        time_t     st_mtime;
        time_t     st_ctime;
        mode_t     st_basemode; /* Permissions with original group perms */
        int64_t    reserved[3];
};
```

For J06.10 and earlier J-series RVUs and H06.21 and earlier H-series RVUs, the **stat** structure uses this definition from the **sys/stat.h** header file:

```
struct  stat64 {
        dev_t      st_dev;
        ino64_t    st_ino;
        mode_t     st_mode;
        nlink_t    st_nlink;
        unsigned int       st_acl:1;
        unsigned int       __filler_1:15;
        uid_t      st_uid;
        gid_t      st_gid;
        dev_t      st_rdev;
        off64_t    st_size;
        time_t     st_atime;
        time_t     st_mtime;
        time_t     st_ctime;
        mode_t     st_basemode; /* Permissions with original group perms */
        int64_t    reserved[3];
};
```

The **fstat64( )** function updates any time-related fields associated with the file before writing into the **stat64** structure, unless it is a read-only fileset. Time-related fields are not updated for read-only OSS filesets.

The fields in the **stat64** structure have these meanings and content:

**st_dev** OSS device identifier for a fileset.

Values for local OSS objects are listed next. Values for local Guardian objects are described in **Use on Guardian Objects**, and values for remote Guardian or OSS objects are described in **Use on Remote Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | ID of device containing directory entry |
| Directory | ID of device containing directory |
| Pipe or FIFO | ID of special fileset for pipes |
| **AF_INET** or **AF_INET6** socket | ID of special fileset for sockets |
| **AF_UNIX** socket | ID of device containing the fileset in which the socket file was created |
| **/dev/null** | ID of device containing directory entry |
| **/dev/tty** | ID of device containing directory entry |

**st_ino** File serial number (inode number). The file serial number and OSS device identifier uniquely identify a regular OSS file within an OSS fileset.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | File serial number (unique) |
| Directory | File serial number (unique) |
| Pipe or FIFO | File serial number (unique) |
| **AF_INET** or **AF_INET6** socket | File serial number (not unique within the HP NonStop node) |
| **AF_UNIX** socket | File serial number of the socket file (unique) |
| **/dev/null** | File serial number (unique) |
| **/dev/tty** | File serial number (unique) |

The **st_ino** value for all node entries in **/E** (including the entry for the logical link from the local node name to the root fileset on the local node) is the value for the root fileset on the corresponding node. If normal conventions are followed, this value is always 0 (zero), so entries in **/E** appear to be nonunique. Values for objects on remote nodes are unique only among the values for objects within the same fileset on that node.

**st_mode** File mode. These bits are ORed into the **st_mode** field:

**S_IFMT** File type. This field can contain one of these values:

**S_IFCHR** Character special file.

**S_IFDIR** Directory.

**S_IFIFO** Pipe or FIFO.

**S_IFREG** Regular file.

**S_IFSOCK** Socket.

For an **AF_INET** or **AF_INET6** socket, the user default permissions are returned for the permission bits. The access flags are set to read and write.

For an **AF_UNIX** socket, the user permissions from the inode for the socket are returned for the permission bits. The access flags are also returned from the inode.

**S_IRWXG** Permissions for the owning group, or if the **st_acl** flag is set, permissions for the the **class** ACL entry.

**S_IRWXO** Other class

**S_IRWXU** Owner class

**S_ISGID** Set group ID on execution

**S_ISUID** Set user ID on execution

**S_ISVTX** Sticky bit; used only for directories (not ORed for files in **/G**, the Guardian file system)

**S_TRUST** Indicates that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers when the memory segment containing the buffers is not shared. This flag applies only to loadfiles for a process, and only a user with appropriate privileges (the super ID) can set it.

**S_TRUSTSHARED**
Indicates that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers regardless of whether the memory segment containing the buffers is shared. This flag applies only to loadfiles for a process, and only a user with appropriate privileges (the super ID) can set it.

Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

**st_nlink** Number of links.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Number of links to the file |
| Directory | Number of links to the directory |
| FIFO | Number of links to the file |
| Pipe | -1 |
| **AF_INET** or **AF_INET6** socket | 0 (zero) |
| **AF_UNIX** socket | Number of links to the socket file |
| **/dev/null** | Number of links to the file |
| **/dev/tty** | Number of links to the file |

**st_acl** If set to 1, indicates that the file has optional access control list (ACL) entries. For compatibility with HP-UX, the member name **st_aclv** is provided as alias for **st_acl**. For more information about ACLs, see the **acl(5)** reference page.

**st_fileprivs** File privileges. For information about file privileges see the **setfilepriv(2)** reference page.

**st_uid** User ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | User ID of the file owner |
| Directory | User ID of the file owner |
| Pipe or FIFO | User ID of the file owner |
| **AF_INET** or **AF_INET6** socket | User ID of the calling process |
| **AF_UNIX** socket | User ID of the creator of the socket file |
| **/dev/null** | User ID of the super ID |
| **/dev/tty** | User ID of the super ID |

**st_gid** Group ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Group ID of the file group |
| Directory | Group ID of the file group |
| Pipe or FIFO | Group ID of the file group |
| **AF_INET** or **AF_INET6** socket | Group ID of the calling process |
| **AF_UNIX** socket | Group ID of the creator of the socket file |
| **/dev/null** | Group ID of the super ID |
| **/dev/tty** | Group ID of the super ID |

**st_basemode** If the **st_acl** flag is set, contains the permissions for the file owner, owning group, and others. If the **st_acl** flag is not set, **st_basemode** is 0 (zero).

**st_rdev** Remote device ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Undefined |
| Directory | Undefined |
| Pipe or FIFO | Undefined |
| **AF_INET** or **AF_INET6** socket | 0 (zero) |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | Undefined |
| **/dev/tty** | ID of the device |

**st_size** File size.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Size of the file in bytes |
| Directory | 4096 |
| Pipe or FIFO | 0 (zero) |
| **AF_INET** or **AF_INET6** socket | 0 (zero) |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | 0 (zero) |
| **/dev/tty** | 0 (zero) |

**st_atime** Access time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last access |
| Directory | Time of the last access |
| Pipe or FIFO | Time of the last access |
| **AF_INET** or **AF_INET6** socket | Value maintained in the socket data structure |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_mtime** Modification time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last data modification |
| Directory | Time of the last modification |
| Pipe or FIFO | Time of the last data modification |
| **AF_INET** or **AF_INET6** socket | Value maintained in the socket data structure |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_ctime** Status change time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Time of the last file status change |
| Directory | Time of the last file status change |
| Pipe or FIFO | Time of the last file status change |
| **AF_INET** or **AF_INET6** socket | Value maintained in the socket data structure |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**Use on Guardian Objects**

The **st_dev** and **st_ino** fields of the **stat64** structure do not uniquely identify Guardian files (files in **/G**).

The **st_dev** field is unique for **/G**, for each disk volume, and for each Telserv process (or other process of subdevice type 30), because each of these is a separate fileset.

The **S_ISGUARDIANOBJECT** macro can indicate whether an object is a Guardian object when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is a Guardian object and **FALSE** otherwise.

The **st_ino** field is a nonunique encoding of the Guardian filename.

The **st_rdev** field contains a unique minor device number for each entry in **/G/ztnt/**, representing each Telserv process subdevice.

The **st_size** field of an EDIT file (file code 101) is the actual (physical) end of file, not the number of bytes in the file. For directories, **st_size** is set to 4096.

When an OSS function is called for a Guardian EDIT file, the **st_mtime** field is set to the last modification time. The **st_atime** field indicates the last time the file was opened, and the **st_ctime** field is set equal to **st_mtime**. No other time-related fields are updated by OSS function calls.

The **st_ctime** and **st_atime** fields for Guardian regular disk files (except for EDIT files) are updated by OSS function calls, not by Guardian procedure calls.

The time fields for **/G**, **/G/***vol*, and **/G/***vol***/***subvol* always contain the current time.

The mapping between Guardian files and their corresponding file types described in the **st_mode** field is listed next:

| Example in /G | Guardian File Type | st_mode File Type | Permissions |
|---|---|---|---|
| **/G** | N/A | Directory | r-xr-xr-x |
| *vol* | Disk volume | Directory | rwxrwxrwx |
| *vol/subvol* | Subvolume | Directory | rwxrwxrwx |
| *vol/subvol/fileid* | Disk file | Regular file | See following text |
| *vol*/**#123** | Temporary disk file | Regular file | See following text |
| **ztnt** | Subtype 30 process | Directory | --x--x--x |
| **ztnt/#pty0001** | Subtype 30 process with qualifier | Character special | rw-rw-rw- |
| **vol1/zyq00001** | Subvolume | Directory | --------- |

A Guardian file classified as a directory is always owned by the super ID.

Guardian permissions are mapped as follows:

- Guardian network or any user permission is mapped to OSS other permission.
- Guardian community or group user permission is mapped to OSS group permission.
- Guardian user or owner permission is mapped to OSS owner permission.
- Guardian super ID permission is OSS super ID permission.
- Guardian read permission is mapped to OSS read permission.
- Guardian write permission is mapped to OSS write permission.
- Guardian execute permission is mapped to OSS execute permission.
- Guardian purge permission is ignored.

Users are not allowed read access to Guardian processes.

OSS file permissions are divided into three groups (owner, group, and other) of three permission bits each (read, write, and execute). The OSS permission bits do not distinguish between remote and local users as Guardian security does; local and remote users are treated alike.

### Use on Remote Objects

The content of the **st_dev** field of the **stat64** structure is unique for each node in **/E** because each node is a separate fileset. Values for directories within **/E** are the same as values for objects on the local HP NonStop node.

The **S_ISEXPANDOBJECT** macro can indicate whether an object in the **/E** directory is on a remote HP NonStop server node when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is on a remote HP NonStop node and **FALSE** otherwise.

### Use From a Threaded Application

This function serializes file operations on an open file. If a thread calls **fstat64( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

## NOTES

For J06.08 and earlier J-series RVUs, H06.19 and earlier H-series RVUs, or G-series RVUs, the OSS Network File System (NFS) cannot access OSS objects that have OSS ACLs that contain optional ACL entries.

For J06.09 and later J-series RVUs and H06.20 and later H-series RVUs, access by the OSS Network File System (NFS) to OSS objects that have OSS ACLs that contain optional ACL entries can be allowed, depending upon the NFSPERMMAP attribute value for the fileset that contains the object. For more information about NFS and ACLs, see the **acl(5)** reference page.

To use the **fstat64( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_fstat64tz(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **fstat64( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not a valid file descriptor.

[EFAULT] The *buffer* parameter points to a location outside of the allocated address space of the process.

[EFSBAD] The program attempted an operation involving a fileset with a corrupted fileset catalog.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOROOT] The program attempted an operation while the root fileset was unavailable.

[ENXIO] An invalid device or address was specified during an input or output operation on a special file. One of these events occurred:

- A device was specified that does not exist, or a request was made beyond the limits of the device.
- The fileset containing the requestor's current working directory or root directory is not mounted. This error can occur after failure and restart of an OSS name server process until the fileset has been repaired and remounted.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation on an input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

**RELATED INFORMATION**

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **chown(2)**, **fstat(2)**, **link(2)**, **mknod(2)**, **open(2)**, **open64(2)**, **pipe(2)**, **spt_fstat64z(2)**, **utime(2)**.

Miscellaneous Topics: **acl(5)**.

**STANDARDS CONFORMANCE**

This function is an HP extension to the XPG4 Version 2 specification.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**fstatvfs -** Gets fileset information for an open file

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/statvfs.h>**

**int fstatvfs(**
    **int** *filedes***,**
    **struct statvfs ****buffer***);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **dup2( )**, **fcntl( )**,**open( )**, or **open64( )** function.

*buffer* Points to a **statvfs** structure that is to hold the returned information for the **fstatvfs( )** call.

## DESCRIPTION

The **fstatvfs( )** function returns descriptive information about a mounted fileset. The information is returned in a **statvfs** structure, which has this definition from the **sys/statvfs.h** header file:

```
typedef struct statvfs {
        unsigned long     f_bsize;
        unsigned long     f_frsize;
        unsigned long     f_blocks;
        unsigned long     f_bfree;
        unsigned long     f_bavail;
        unsigned long     f_files;
        unsigned long     f_ffree;
        unsigned long     f_favail;
        unsigned long     f_fsid;
        char              f_basetype[FSTYPSZ];
        unsigned long     f_flag;
        unsigned long     f_namemax;
        char              f_fstr[32];
        unsigned long     f_bminavail;
        unsigned long     f_bmaxavail;
        unsigned long     f_filler[5];
} statvfs_t;
```

The fields in this structure have these meanings and content:

**f_bsize** Fileset block size:

| For | Contains |
|---|---|
| Regular file | 4096 |
| Directory | 4096 |
| FIFO | 4096 |
| **/dev/null** | 4096 |
| Object in **/G** | 4096 |
| **/G** | 4096 |
| Terminal device file | 4096 |
| **/E** | 4096 |

**f_frsize** Fundamental file system block size:

| For | Contains |
|---|---|
| Regular file | 4096 |
| Directory | 4096 |
| FIFO | 4096 |
| **/dev/null** | 4096 |
| Object in **/G** | 4096 |
| **/G** | 4096 |
| Terminal device file | 4096 |
| **/E** | 4096 |

**f_blocks** Total number of blocks in fileset, in units of **f_frsize**:

| For | Contains |
|---|---|
| Regular file | Number of blocks on all volumes ever used in the fileset. |
| Directory | Number of blocks on all volumes ever used in the fileset. |
| FIFO | Number of blocks on all volumes ever used in the fileset. |
| **/dev/null** | Number of blocks on all volumes ever used in the fileset. |
| Object in **/G** | Number of blocks on the volume containing the object. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_bfree** Total number of free blocks in fileset:

| **For** | **Contains** |
|---|---|
| Regular file | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Directory | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| FIFO | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| **/dev/null** | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Object in **/G** | Number of free blocks in the volume containing the object. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_bavail** Number of free blocks available to a process without appropriate privileges:

| **For** | **Contains** |
|---|---|
| Regular file | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Directory | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| FIFO | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| **/dev/null** | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Object in **/G** | Number of free blocks in the volume containing the object. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_files** Total number of file serial numbers (inode numbers) in the fileset:

| For | Contains |
|---|---|
| Regular file | Number of inode numbers in the fileset. |
| Directory | Number of inode numbers in the fileset. |
| FIFO | Number of inode numbers in the fileset. |
| **/dev/null** | Number of inode numbers in the fileset. |
| Object in **/G** | The value of **ULONG_MAX**. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_ffree** Total number of free file serial numbers (inode numbers) in the fileset:

| For | Contains |
|---|---|
| Regular file | Number of free inode numbers in the fileset. |
| Directory | Number of free inode numbers in the fileset. |
| FIFO | Number of free inode numbers in the fileset. |
| **/dev/null** | Number of free inode numbers in the fileset. |
| Object in **/G** | The value of **ULONG_MAX**. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_favail** Number of file serial numbers (inode numbers) available to a process without appropriate privileges:

| For | Contains |
|---|---|
| Regular file | Number of free inode numbers in the fileset. |
| Directory | Number of free inode numbers in the fileset. |
| FIFO | Number of free inode numbers in the fileset. |
| **/dev/null** | Number of free inode numbers in the fileset. |
| Object in **/G** | The value of **ULONG_MAX**. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_fsid** Fileset identifier:

| **For** | **Contains** |
|---|---|
| Regular file | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| Directory | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| FIFO | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/dev/null** | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| Object in **/G** | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/G** | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| Terminal device file | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/E** | Lower 32 bits of the **st_dev** field in the **stat** structure. |

**f_basetype** Type of file system:

| **For** | **Contains** |
|---|---|
| Regular file | **OSS** |
| Directory | **OSS** |
| FIFO | **OSS** |
| **/dev/null** | **OSS** |
| Object in **/G** | **GUARDIAN** |
| **/G** | **GUARDIAN** |
| Terminal device file | **GUARDIAN** |
| **/E** | **EXPAND** |

**f_flag** Bit mask indicating type of fileset access allowed:

| **For** | **Contains** |
|---|---|
| Regular file | 4 if fileset is read/write, 5 if fileset is read-only. |
| Directory | 4 if fileset is read/write, 5 if fileset is read-only. |
| FIFO | 4 if fileset is read/write, 5 if fileset is read-only. |
| **/dev/null** | 4 if fileset is read/write, 5 if fileset is read-only. |
| Object in **/G** | 2 |
| **/G** | 3 |

| | |
|---|---|
| Terminal device file | 2 |
| **/E** | 3 |

The content of the **f_flag** field can be tested with these symbolic values:

**ST_NOSUID** This bit flag is set if the fileset does not allow the **setuid** bit to be set for its member files.

**ST_NOTRUNC**
This bit flag is set if the fileset does not truncate filenames.

**ST_RDONLY** This bit flag is set if the fileset is mounted for read-only access.

**f_namemax** Maximum number of character bytes in a filename within the fileset:

| **For** | **Contains** |
|---|---|
| Regular file | 248 |
| Directory | 248 |
| FIFO | 248 |
| **/dev/null** | 248 |
| Object in **/G** | 8 |
| **/G** | 7 |
| Terminal device file | 7 |
| **/E** | 7 |

**f_fstr** Fileset pathname prefix string:

| **For** | **Contains** |
|---|---|
| Regular file | **/E**/*nodename*/**G**/*volume*/**ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| Directory | **/E**/*nodename*/**G**/*volume*/**ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| FIFO | **/E**/*nodename*/**G**/*volume*/**ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| **/dev/null** | **/E**/*nodename*/**G**/*volume*/**ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| Object in **/G** | **/E**/*nodename*/**G**/*volume*, identifying the disk volume containing the specified file. |
| **/G** | **/E**/*nodename*/**G** |
| Terminal device file | **/E**/*nodename*/**G** |
| **/E** | **/E** |

**f_bminavail** Number of blocks free on the disk volume with the least space remaining:

| For | Contains |
|---|---|
| Regular file | Number of blocks. |
| Directory | Number of blocks. |
| FIFO | Number of blocks. |
| **/dev/null** | Number of blocks. |
| Object in **/G** | Number of blocks. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_bmaxavail** Number of blocks free on the disk volume with the most space remaining:

| For | Contains |
|---|---|
| Regular file | Number of blocks. |
| Directory | Number of blocks. |
| FIFO | Number of blocks. |
| **/dev/null** | Number of blocks. |
| Object in **/G** | Number of blocks. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

## NOTES

This function provides compatibility with the System V Interface Definition, Revision 3.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **fstatvfs( )** function returns the value 0 (zero). Otherwise, it returns the value -1, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **fstatvfs( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not a valid file descriptor.

[EFAULT] The *buffer* parameter points to an invalid address.

[EINTR] The function was interrupted by a signal before any data arrived.

[EINVAL] The file pointed to by the *filedes* parameter is an OSS pipe or a socket.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
- A physical I/O error occurred. The device holding the file might be in the down state, or both processors that provide access to the device

might have failed. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[EOVERFLOW]
One of the values returned cannot be represented correctly in the structure pointed to by the *buffer* **parameter.**

**[EWRONGID]**
One of these conditions occurred:

- The process attempted an input or output operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for use of the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **fstat(2)**, **fstat64(2), lstat(2)**, **lstat64(2)**, **stat(2)**, **stat64(2)**, **statvfs(2)**, **statvfs64(2)**.

## STANDARDS CONFORMANCE

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EINVAL], [EISGUARDIAN], [ENETDOWN], and [EWRONGID] can be returned.

## NAME

**fstatvfs64 -** Gets fileset information for an open file

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/statvfs.h>**

**int fstatvfs64(**
  **int** *filedes***,**
  **struct statvfs64 ***buffer***);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, or **open64( )** function.

*buffer* Points to a **statvfs64** structure that is to hold the returned information for the **fstatvfs64( )** call.

## DESCRIPTION

The **fstatvfs64( )** function is similar to the **fstatvfs( )** function except that, in addition to supporting smaller files, the **fstatvfs64( )** function supports OSS files larger than approximately 2 gigabytes.

An application can explicitly call this function when the application is compiled using the **#define _LARGEFILE64_SOURCE 1** feature test macro or an equivalent compiler command option.

An application call to **fstatvfs( )** is automatically mapped to this function when the applciation is compiled using the **#define _FILE_OFFSET_BITS 64** feature test macro or an equivalent compiler command option.

The **fstatvfs64( )** function returns descriptive information about a mounted fileset. The information is returned in a **statvfs64** structure, which has this definition from the **sys/statvfs.h** header file:

```
typedef struct statvfs64 {
        u_long          f_bsize;
        u_long          f_frsize;
        fsblkcnt64_t    f_blocks;
        fsblkcnt64_t    f_bfree;
        fsblkcnt64_t    f_bavail;
        fsfilcnt64_t    f_files;
        fsfilcnt64_t    f_ffree;
        fsfilcnt64_t    f_favail;
        u_long          f_fsid;
        char            f_basetype[FSTYPSZ];
        u_long          f_flag;
        u_long          f_namemax;
        char            f_fstr[32];
        fsblkcnt64_t    f_bminavail;
        fsblkcnt64_t    f_bmaxavail;
        u_long          f_filler[5];
} statvfs64_t;
```

The fields in this structure have these meanings and content:

**f_bsize** Fileset block size:

| For | Contains |
|---|---|
| Regular file | 4096 |
| Directory | 4096 |
| FIFO | 4096 |
| **/dev/null** | 4096 |
| Object in **/G** | 4096 |
| **/G** | 4096 |
| Terminal device file | 4096 |
| **/E** | 4096 |

**f_frsize** Fundamental file system block size:

| For | Contains |
|---|---|
| Regular file | 4096 |
| Directory | 4096 |
| FIFO | 4096 |
| **/dev/null** | 4096 |
| Object in **/G** | 4096 |
| **/G** | 4096 |
| Terminal device file | 4096 |
| **/E** | 4096 |

**f_blocks** Total number of blocks in fileset, in units of **f_frsize**:

| For | Contains |
|---|---|
| Regular file | Number of blocks on all volumes ever used in the fileset. |
| Directory | Number of blocks on all volumes ever used in the fileset. |
| FIFO | Number of blocks on all volumes ever used in the fileset. |
| **/dev/null** | Number of blocks on all volumes ever used in the fileset. |
| Object in **/G** | Number of blocks on the volume containing the object. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_bfree** Total number of free blocks in fileset:

| **For** | **Contains** |
|---|---|
| Regular file | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Directory | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| FIFO | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| **/dev/null** | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Object in **/G** | Number of free blocks in the volume containing the object. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_bavail** Number of free blocks available to a process without appropriate privileges:

| **For** | **Contains** |
|---|---|
| Regular file | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Directory | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| FIFO | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| **/dev/null** | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Object in **/G** | Number of free blocks in the volume containing the object. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_files** Total number of file serial numbers (inode numbers) in the fileset:

| For | Contains |
|---|---|
| Regular file | Number of inode numbers in the fileset. |
| Directory | Number of inode numbers in the fileset. |
| FIFO | Number of inode numbers in the fileset. |
| **/dev/null** | Number of inode numbers in the fileset. |
| Object in **/G** | The value of **ULONG_MAX**. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_ffree** Total number of free file serial numbers (inode numbers) in the fileset:

| For | Contains |
|---|---|
| Regular file | Number of free inode numbers in the fileset. |
| Directory | Number of free inode numbers in the fileset. |
| FIFO | Number of free inode numbers in the fileset. |
| **/dev/null** | Number of free inode numbers in the fileset. |
| Object in **/G** | The value of **ULONG_MAX**. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_favail** Number of file serial numbers (inode numbers) available to a process without appropriate privileges:

| For | Contains |
|---|---|
| Regular file | Number of free inode numbers in the fileset. |
| Directory | Number of free inode numbers in the fileset. |
| FIFO | Number of free inode numbers in the fileset. |
| **/dev/null** | Number of free inode numbers in the fileset. |
| Object in **/G** | The value of **ULONG_MAX**. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_fsid** Fileset identifier:

| **For** | **Contains** |
|---|---|
| Regular file | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| Directory | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| FIFO | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/dev/null** | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| Object in **/G** | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/G** | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| Terminal device file | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/E** | Lower 32 bits of the **st_dev** field in the **stat** structure. |

**f_basetype** Type of file system:

| **For** | **Contains** |
|---|---|
| Regular file | **OSS** |
| Directory | **OSS** |
| FIFO | **OSS** |
| **/dev/null** | **OSS** |
| Object in **/G** | **GUARDIAN** |
| **/G** | **GUARDIAN** |
| Terminal device file | **GUARDIAN** |
| **/E** | **EXPAND** |

**f_flag** Bit mask indicating type of fileset access allowed:

| **For** | **Contains** |
|---|---|
| Regular file | 4 if fileset is read/write, 5 if fileset is read-only. |
| Directory | 4 if fileset is read/write, 5 if fileset is read-only. |
| FIFO | 4 if fileset is read/write, 5 if fileset is read-only. |
| **/dev/null** | 4 if fileset is read/write, 5 if fileset is read-only. |
| Object in **/G** | 2 |
| **/G** | 3 |

| | |
|---|---|
| Terminal device file | 2 |
| **/E** | 3 |

The content of the **f_flag** field can be tested with these symbolic values:

**ST_NOSUID** This bit flag is set if the fileset does not allow the **setuid** bit to be set for its member files.

**ST_NOTRUNC**
This bit flag is set if the fileset does not truncate filenames.

**ST_RDONLY** This bit flag is set if the fileset is mounted for read-only access.

**f_namemax** Maximum number of character bytes in a filename within the fileset:

| **For** | **Contains** |
|---|---|
| Regular file | 248 |
| Directory | 248 |
| FIFO | 248 |
| **/dev/null** | 248 |
| Object in **/G** | 8 |
| **/G** | 7 |
| Terminal device file | 7 |
| **/E** | 7 |

**f_fstr** Fileset pathname prefix string:

| **For** | **Contains** |
|---|---|
| Regular file | **/E**/*nodename*/**G**/*volume*/**ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| Directory | **/E**/*nodename*/**G**/*volume*/**ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| FIFO | **/E**/*nodename*/**G**/*volume*/**ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| **/dev/null** | **/E**/*nodename*/**G**/*volume*/**ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| Object in **/G** | **/E**/*nodename*/**G**/*volume*, identifying the disk volume containing the specified file. |
| **/G** | **/E**/*nodename*/**G** |
| Terminal device file | **/E**/*nodename*/**G** |
| **/E** | **/E** |

**f_bminavail** Number of blocks free on the disk volume with the least space remaining:

| **For** | **Contains** |
|---|---|
| Regular file | Number of blocks. |
| Directory | Number of blocks. |
| FIFO | Number of blocks. |
| **/dev/null** | Number of blocks. |
| Object in **/G** | Number of blocks. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

**f_bmaxavail** Number of blocks free on the disk volume with the most space remaining:

| **For** | **Contains** |
|---|---|
| Regular file | Number of blocks. |
| Directory | Number of blocks. |
| FIFO | Number of blocks. |
| **/dev/null** | Number of blocks. |
| Object in **/G** | Number of blocks. |
| **/G** | 0 |
| Terminal device file | 0 |
| **/E** | 0 |

## NOTES

This function provides compatibility with the System V Interface Definition, Revision 3.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **fstatvfs64( )** function returns the value 0 (zero). Otherwise, it returns the value -1, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **fstatvfs64( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not a valid file descriptor.

[EFAULT] The *buffer* parameter points to an invalid address.

[EINTR] The function was interrupted by a signal before any data arrived.

[EINVAL] The file pointed to by the *filedes* parameter is an OSS pipe or a socket.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
- A physical I/O error occurred. The device holding the file might be in the down state, or both processors that provide access to the device

might have failed. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[EWRONGID] One of these conditions occurred:

- The process attempted an input or output operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for use of the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **fstat(2)**, **fstat64(2)**, **lstat(2)**, **lstat64(2)**, **stat(2)**, **stat64(2)**, **statvfs(2)** **statvfs64(2)**.

## STANDARDS CONFORMANCE

This function is an HP extension to the XPG4 Version 2 specification.

## NAME

**fsync -** Writes modified data and file attributes to permanent storage

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <unistd.h>

int fsync(
        int filedes);
```

## PARAMETERS

*filedes*
: Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

  When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function, or the thread-aware **accept( )**, **dup2( )**, or **fcntl( )** function.

## DESCRIPTION

The **fsync( )** function saves all modifications for the file open specified by the *filedes* parameter. On return from the **fsync( )** function, all updated data and file attributes have been saved on permanent storage.

### Use on Guardian Objects

The *filedes* parameter can specify any regular file in **/G** including Guardian EDIT files. Time values are not saved for other file types in **/G**, such as terminal files.

### Use From a Threaded Application

If this function must wait for an I/O operation to complete on an open file, this function blocks the thread (instead of the entire process) that called it, while it waits for the I/O operation to complete.

## NOTES

The **fsync( )** function offers an alternative to the **O_SYNC** file status flag. Using **fsync( )** calls gives an application control over the performance trade offs involved in guaranteeing data integrity. OSS file-system caching can be used for files that are protected only by **fsync( )** function calls.

To use the **fsync( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_fsyncz(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **fsync( )** function returns the value 0 (zero). Otherwise, it returns the value -1, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **fsync( )** function sets **errno** to the value that corresponds to the condition:

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function is in progress on a regular file and a function that is process-blocking for regular files attempts to begin an I/O operation on the same open file.

If the **fsync( )** function is thread-aware, the [EALREADY] value is not returned.

[EBADF] The *filedes* parameter is not a valid file descriptor.

[EINTR] The **fsync( )** function was interrupted by a signal that was caught.

[EINVAL] The *filedes* parameter, although valid, does not refer to a file on which this operation is possible.

[EIO] An I/O error occurred during a write to the fileset.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENXIO] No such device or address. An invalid device or address was specified during an input or output operation on a special file. One of these events occurred:

- A device was specified that does not exist, or a request was made beyond the limits of the device.

- The fileset containing the requestor’s current working directory or root directory is not mounted. This error can occur after failure and restart of an OSS name server process until the fileset has been repaired and remounted.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **open(2)**, **socket(2)**, **spt_fsynchz(2)**, **stat(2)**, **write(2)**.

## STANDARDS CONFORMANCE

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EISGUARDIAN], [ENETDOWN], [ENXIO], and [EWRONGID] can be returned.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**ftruncate -** Changes file length

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <sys/types.h>**

**int ftruncate(**
    **int** *filedes***,**
    **off_t** *length***);**

## PARAMETERS

*filedes* Specifies the descriptor of a file that must be open for writing.

When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, or **open64( )** function, or the thread-aware **dup2( )** or **fcntl( )** function.

*length* Specifies the new length of the file in bytes.

## DESCRIPTION

The **ftruncate( )** function changes the length of a file to the size, in bytes, specified by the *length* parameter.

If the new length is less than the previous length, the **ftruncate( )** function removes all data beyond *length* bytes from the specified file. All file data between the new EOF and the previous EOF is discarded.

If the new length is greater than the previous length, zeros are added between the previous EOF and the new EOF.

Full blocks are returned to the fileset so that they can be used again, and the file size is changed to the value of the *length* parameter.

The **ftruncate( )** function has no effect on First-in, First-out (FIFO) special files. This function does not modify the seek pointer of the file. If **ftruncate( )** is called for a FIFO file, the call fails, and **errno** is set to [EINVAL].

Upon successful completion, the **ftruncate( )** function marks the **st_ctime** and **st_mtime** fields of the file for update. If the file is a regular file, the **ftruncate( )** function clears the **S_ISUID** and **S_ISGID** attributes of the file.

### Use From a Threaded Application

The thread-aware **ftruncate( )** function offers an alternative to the **O_SYNC** file status flag. Using thread-aware **ftruncate( )** gives a threaded application control over the performance trade offs involved in guaranteeing data integrity. OSS file-system caching can be used for files that are protected only by thread-aware **ftruncate( )** function calls.

If this function must wait for an I/O operation to complete on an open file, this function blocks the thread (instead of the entire process) that called it, while it waits for the I/O operation to complete.

## NOTES

To use the **ftruncate( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_ftruncatez(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **ftruncate( )** function sets **errno** to the corresponding value:

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function is in progress on a regular file and a function that is process-blocking for regular files attempts to begin an I/O operation on the same open file.

If the **ftruncate( )** function is thread-aware, the [EALREADY] value is not returned.

[EBADF] The *filedes* parameter does not specify a valid file descriptor open for writing.

[EFBIG] The *length* parameter is greater than the minimum of 2 gigabytes minus 1 byte and the maximum file size established during file open.

[EINTR] The function was interrupted by a signal before any data arrived.

[EINVAL] One of these conditions occurred:

- The file pointed to by the *filedes* parameter is not a regular file.
- The value specified for the *length* parameter was less than 0 (zero).

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to read from its controlling terminal, the process is ignoring or blocking the **SIGTTIN** signal, or the process group is orphaned.
- A physical I/O error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[EROFS] The file resides on a read-only fileset.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **chmod(2)**, **fcntl(2)**, **ftruncate64(2)**, **open(2)**, **open64(2)**, **spt_ftruncatez(2)**.

## STANDARDS CONFORMANCE

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EISGUARDIAN], [ENETDOWN], and [EROFS] can be returned.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

This function cannot be used as a cancellation point when the function is used with the POSIX User Thread Model library.

## NAME

**ftruncate64 -** Changes file length

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <sys/types.h>**

**int ftruncate64(**
**int** *filedes***,**
**off64_t** *length***);**

## PARAMETERS

*filedes* Specifies the descriptor of a file that must be open for writing.

When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, or **open64( )** function, or the thread-aware **dup2( )** or **fcntl( )** function.

*length* Specifies the new length of the file in bytes.

## DESCRIPTION

The **ftruncate64( )** function is similar to the **ftruncate( )** function except that, in addition to supporting smaller files, the **ftruncate64( )** function supports OSS files larger than approximately 2 gigabytes.

An application can explicitly call this function when the application is compiled using the **#define _LARGEFILE64_SOURCE 1** feature test macro or an equivalent compiler command option.

An application call to **ftruncate( )** is automatically mapped to this function when the applciation is compiled using the **#define _FILE_OFFSET_BITS 64** feature test macro or an equivalent compiler command option.

The **ftruncate64( )** function changes the length of a file to the size, in bytes, specified by the *length* parameter.

If the new length is less than the previous length, the **ftruncate64( )** function removes all data beyond *length* bytes from the specified file. All file data between the new EOF and the previous EOF is discarded.

If the new length is greater than the previous length, zeros are added between the previous EOF and the new EOF. If the new length would exceed the file size limit for the calling process, the call to **ftruncate64( )** fails, and **errno** is set to [EFBIG].

Full blocks are returned to the fileset so that they can be used again, and the file size is changed to the value of the *length* parameter.

The **ftruncate64( )** function has no effect on First-in, First out (FIFO) special files. This function does not modify the seek pointer of the file. If **ftruncate64( )** is called for a FIFO file, the call fails, and **errno** is set to [EINVAL].

Upon successful completion, the **ftruncate64( )** function marks the **st_ctime** and **st_mtime** fields of the file for update. If the file is a regular file, the **ftruncate64( )** function clears the **S_ISUID** and **S_ISGID** attributes of the file.

### Use From a Threaded Application

The thread-aware **ftruncate64( )** function offers an alternative to the **O_SYNC** file status flag. Using thread-aware **ftruncate64( )** gives a threaded application control over the performance trade offs involved in guaranteeing data integrity. OSS file-system caching can be used for files that are protected only by thread-aware **ftruncate64( )** function calls.

If this function must wait for an I/O operation to complete on an open file, this function blocks the thread (instead of the entire process) that called it, while it waits for the I/O operation to complete.

## NOTES

To use the **ftruncate64( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_ftruncate64z(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **ftruncate64( )** function sets **errno** to the corresponding value:

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function is in progress on a regular file and a function that is process-blocking for regular files attempts to begin an I/O operation on the same open file.

If the **ftruncate64( )** function is thread-aware, the [EALREADY] value is not returned.

[EBADF] The *filedes* parameter does not specify a valid file descriptor open for writing.

[EFBIG] The *length* parameter is greater than the minimum of 2 gigabytes minus 1 byte and the maximum file size established during file open.

[EINTR] The function was interrupted by a signal before any data arrived.

[EINVAL] One of these conditions occurred:

- The file pointed to by the *filedes* parameter is not a regular file.
- The value specified for the *length* parameter was less than 0 (zero).

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to read from its controlling terminal, the process is ignoring or blocking the **SIGTTIN** signal, or the process group is orphaned.
- A physical I/O error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[EROFS] The file resides on a read-only fileset.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

### RELATED INFORMATION

Functions: **chmod(2)**, **fcntl(2)**, **open(2)**, **open64(2)**, **spt_ftruncate64z(2)**.

### STANDARDS CONFORMANCE

This function is an HP extension to the XPG4 Version 2 specification.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

This function cannot be used as a cancellation point when the function is used with the POSIX User Thread Model library.

## NAME

**getegid -** Gets the effective group ID

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsecsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zsecdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/ysecdll**

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>

gid_t getegid(void);
```

## DESCRIPTION

The **getegid( )** function returns the effective group ID of the calling process.

The real, effective, and saved group IDs are set at login time and when a process with appropriate privileges calls the **setgid( )** function. The effective and saved group IDs can also change as a result of executing a set-group-ID program.

## NOTES

A process's effective group ID can be different from the group membership indicated by the operating system process access ID (PAID) of the process.

## RETURN VALUES

The **getegid( )** function returns the effective group ID. It is always successful.

The "uninitialized" group ID (hexadecimal 80000000) is returned when authentication information of a process is uninitialized.

## RELATED INFORMATION

Functions: **getgid(2)**, **getgroups(2)**, **setgid(2)**.

Commands: **id(1)**.

**NAME**

**geteuid -** Gets the effective user ID of the current process

**LIBRARY**

G-series native OSS processes: **/G/system/sys***nn***/zsecsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zsecdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/ysecdll**

**SYNOPSIS**

```
#include <sys/types.h>      /* optional except for POSIX.1 */
#include <unistd.h>

uid_t geteuid(void);
```

**DESCRIPTION**

The **geteuid( )** function returns the effective user ID of the current process.

**RETURN VALUES**

The **geteuid( )** function returns the requested user ID. It is always successful.

When the authentication information for a process is uninitialized, the "uninitialized" user ID (hexadecimal 80000000) is returned.

**RELATED INFORMATION**

Functions: **getuid(2)**, **setuid(2)**.

## NAME

**getgid -** Gets the real group ID

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsecsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zsecdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/ysecdll**

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>

gid_t getgid(void);
```

## DESCRIPTION

The **getgid( )** function returns the real group ID of the calling process.

The real, effective, and saved group IDs are set at login time and when a process with appropriate privileges calls the **setgid( )** function.

## NOTES

A process's real group ID can be different from the group number in the operating system creator access ID (CAID) for a process.

## RETURN VALUES

The **getgid( )** function returns the real group ID. It is always successful.

The "uninitialized" group ID (hexadecimal 80000000) is returned when authentication information of a process is uninitialized.

## RELATED INFORMATION

Functions: **getegid(2)**, **getgroups(2)**, **setgid(2)**.

Commands: **id(1)**.

## NAME

**getgroups -** Gets the group list of the current process

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsecsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zsecdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/ysecdll**

## SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>

int getgroups(
        int gidsetsize,
        gid_t grouplist [ ]);
```

## PARAMETERS

*gidsetsize* Specifies the number of entries that can be stored in the array pointed to by the *grouplist* parameter.

*grouplist* Points to the array in which the group list of the process is stored.

## DESCRIPTION

The **getgroups( )** function gets the group list of the current process. The list is stored in the array pointed to by the *grouplist* parameter. The *gidsetsize* parameter indicates the number of entries that can be stored in this array.

The **getgroups( )** function never returns more than **NGROUPS_MAX** entries. (**NGROUPS_MAX** is a constant defined in the **limits.h** header file.) If the *gidsetsize* parameter is 0 (zero), the **getgroups( )** function returns the number of groups in the group list.

The effective group ID may not occur in the returned group ID list if the effective group ID has been changed by executing a set-group-ID program or by calling the **setgid( )** function.

## RETURN VALUES

Upon successful completion, the **getgroups( )** function returns the number of elements stored in the array pointed to by the *grouplist* parameter. If **getgroups( )** fails, then the value -1 is returned and **errno** is set to indicate the error.

The value 0 (zero) is returned when the authentication information for a process is uninitialized.

## ERRORS

If any of these conditions occur, the **getgroups( )** function sets **errno** to the corresponding value:

[EFAULT] The *gidsetsize* and *grouplist* parameters specify an array that is partially or completely outside the allocated address space of the process.

[EINVAL] The *gidsetsize* parameter is nonzero and smaller than the number of groups in the group list, or the *grouplist* parameter is out of range.

## RELATED INFORMATION

Commands: **id(1)**.

## NAME

**gethostname -** Gets the name of the local host

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zinetsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zinetdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yinetdll**

## SYNOPSIS

**#include <netdb.h>**

**int gethostname**(
    **char** ***address*,
    **int** *address_len***);**

## PARAMETERS

*address* Returns the address of an array of bytes where the hostname is stored. If sufficient space is provided, the returned *address* parameter is NULL terminated.

*address_len* Specifies the length of the array pointed to by the *address* parameter.

## DESCRIPTION

The **gethostname( )** function retrieves the standard hostname of the local host. The name returned corresponds to the hostname returned in the Subsystem Command Facility (SCF) command INFO PROCESS for the TCP subsystem.

## RETURN VALUES

Upon successful completion, the **gethostname( )** function returns a value of 0 (zero). Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occur, the **gethostname( )** function sets **errno** to the corresponding value:

[EINVAL] The *address* parameter or *address_len* parameter refers to an invalid address in the user's address space.

## RELATED INFORMATION

Functions: **gethostid(2)**.

## NAME

**getpeername -** Gets the name of the peer socket

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int getpeername(
        int socket,
        struct sockaddr *address,
        socklen_t *address_len);
```

## PARAMETERS

*socket* Specifies the open file descriptor of the socket.

*address* Points to a **sockaddr** structure, where the address of the peer socket is returned.

If the length of the address is greater than the length of the supplied **sockaddr** structure, the address is truncated when stored.

If the protocol permits connection by unbound clients, and if the peer socket is not bound to an address, then the value stored is unspecified.

The length and format of the address depend on the address family of the socket.

For **AF_INET** sockets, a pointer to the address structure **sockaddr_in** can be cast as a **struct sockaddr**. For **AF_INET6** sockets, a pointer to the address structure **sockaddr_in6** can be cast as a **struct sockaddr**. For **AF_UNIX** sockets, a pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*address_len* Points to a **socklen_t** data item, which, on input, specifies the length of the **sockaddr** structure pointed to by the *address* parameter, and, on output, specifies the length of the address returned.

## DESCRIPTION

The **getpeername( )** function retrieves the peer address of the specified socket, stores this address in the **sockaddr** structure pointed to by the *address* parameter, and stores the length of this address in the object pointed to by the *address_len* parameter.

The behavior of this function differs depending on the type of socket and whether the socket is an **AF_UNIX** Release 1 socket or an **AF_UNIX** Release 2 socket in compatibility mode, or the socket is an **AF_UNIX** Release 2 socket in portability mode.

| **AF_UNIX Release 1 or AF_UNIX Release 2 in Compatibility Mode** | **AF_UNIX Release 2 in Portability Mode** |
|---|---|
| After a successful call to **connect( )** on a stream socket where a relative pathname is passed in the *address* parameter, a subsequent call to **getpeername( )** returns the relative path name passed to **connect( )**. | After a successful call to **connect( )** on a stream socket where a relative pathname is passed in the *address* parameter, a subsequent call to **getpeername( )** returns the fully-qualified equivalent of the relative path name passed to **connect( )**. |
| After a successful call to **connect( )** on a datagram socket, when the file specified in the *address* parameter is renamed, a subsequent call to **getpeername( )** fails with *errno* set to [ENOENT]. | After a successful call to **connect( )** on a datagram socket, when the file specified in the *address* parameter is renamed, a subsequent call to **getpeername( )** returns the name of the file at the time of the **connect( )** call. |
| After a successful call to **connect( )** on a datagram socket, if the file specified in the *address* parameter is unlinked, a subsequent call to **getpeername( )** fails with *errno* set to [ENOENT]. | After a successful call to **connect( )** on a datagram socket, if the file specified in the *address* parameter is unlinked, a subsequent call to **getpeername( )** returns the fully-qualified name of the file passed to the **connect( )** call. |

For more information about **AF_UNIX** Release 2 sockets, portability mode, and compatibility mode, see the *Open System Services Programmer's Guide*.

**NOTES**

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

A process can use the **getsockname( )** function to retrieve the locally bound name of a socket.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

**RETURN VALUES**

Upon successful completion, the **getpeername( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **getpeername( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.

- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINVAL] The socket has been shut down.

[ENOBUFS] There were not enough system resources available to complete the call. A retry at a later time might succeed.

[ENOENT] The socket is either an **AF_UNIX** Release 1 datagram socket or an **AF_UNIX** Release 2 datagram socket in compatibility mode and one of these conditions occurred:

- The underlying file of the peer socket specified in the *address* parameter was renamed.
- The underlying file of the peer socket specfied in the *address* parameter was unlinked.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOTCONN] The socket is not connected or has not had the peer socket previously specified.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The operation is not supported for the protocol of the socket specified by the *socket* parameter.

**RELATED INFORMATION**

Functions: **accept(2)**, **bind(2)**, **getsockname(2)**, **socket(2)**.

**STANDARDS CONFORMANCE**

The HP implementation does not return the **errno** value [ENOSR].

The following are HP extensions to the XPG4 specification:

- The **errno** value [ECONNRESET] can be returned when the transport-provider process is unavailable.

**NAME**

**getpgid -** Gets the process group ID for a specified OSS process

**LIBRARY**

G-series native OSS processes: **/G/system/sys*nn*/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/yosskdll**

**SYNOPSIS**

**#include <unistd.h>**

**pid_t getpgid(**
    **pid_t** *pid***);**

**PARAMETERS**

*pid* Specifies the process ID of the target process. If the specified value is 0 (zero), the target process is the calling process.

**DESCRIPTION**

The **getpgid( )** function returns the process group ID of the process specified by the process ID *pid*.

**Use From the Guardian Environment**

This function cannot be used in the Guardian environment. When the **getpgid( )** function is called from the Guardian environment, the call fails and **errno** is set to [ENOTOSS].

**RETURN VALUES**

The **getpgid( )** function returns the process group ID of the specified process. If an error occurs, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **getpgid( )** function sets **errno** to the corresponding value.

[EINVAL] The *pid* parameter is out of range.

[ENOTOSS] The calling process is not an OSS process. The requested operation is not supported from the Guardian environment.

[EPERM] The specified process is not in the same session as the calling process.

[ESRCH] The value of the *pid* parameter does not match the OSS process ID of the calling process or of a child process of the calling process.

**RELATED INFORMATION**

Functions: **exec(2)**, **fork(2)**, **getpgrp(2)**, **setpgid(2)**, **setsid(2)**, **tcsetpgrp(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**.

**STANDARDS CONFORMANCE**

The following is an HP extension to the XPG4 Version 2 specification:

- The function can return the **errno** value [ENOTOSS].

## NAME

**getpgrp -** Gets the process group ID of the calling process

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>

pid_t getpgrp(void);
```

## DESCRIPTION

The **getpgrp( )** function returns the process group ID of the calling process.

### Use From the Guardian Environment

Calls to **getpgrp( )** from Guardian processes are not successful.

## RETURN VALUES

When called by an OSS process, this function returns the requested process group ID. When called by a Guardian process, Guardian trap number 5 is set.

## RELATED INFORMATION

Commands: **ps(1)**.

Functions: **exec(2)**, **_exit(2)**, **fork(2)**, **kill(2)**, **setpgid(2)**, **setsid(2)**, **tdm_fork(2)**.

## NAME

**getpid -** Gets the OSS process ID

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>

pid_t getpid(void);
```

## DESCRIPTION

The **getpid( )** function returns the OSS process ID of the calling process.

### Use From the Guardian Environment

Calls to **getpid( )** from Guardian processes are not successful.

## RETURN VALUES

When called by an OSS process, this function returns the requested OSS process ID. When called by a Guardian process, Guardian trap number 5 is set.

## RELATED INFORMATION

Commands: **ps(1)**.

Functions: **exec(2)**, **_exit(2)**, **fork(2)**, **kill(2)**, **setpgid(2)**, **setsid(2)**, **tdm_fork(2)**.

## NAME

**getppid -** Gets the parent OSS process ID

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>

pid_t getppid(void);
```

## DESCRIPTION

The **getppid( )** function returns the parent OSS process ID of the calling process. If the parent process terminates or the calling process was created by a Guardian process, **getppid( )** returns a parent OSS process ID of 1.

### Use From the Guardian Environment

Calls to **getppid( )** from Guardian processes are not successful.

## RETURN VALUES

When called by an OSS process, this function returns the requested OSS process ID. When called by a Guardian process, Guardian trap number 5 is set.

## RELATED INFORMATION

Commands: **ps(1)**.

Functions: **exec(2)**, **_exit(2)**, **fork(2)**, **kill(2)**, **setpgid(2)**, **setsid(2)**, **tdm_fork(2)**.

**NAME**

**getpriority -** Gets the OSS process scheduling priority

**LIBRARY**

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

**SYNOPSIS**

```
#include <sys/resource.h>

int getpriority(
        int which,
        id_t who);
```

**PARAMETERS**

*which* Specifies the symbolic value for the source of the **nice** value to be returned. The following symbolic values defined in the **sys/resource.h** header file are valid:

**PRIO_PGRP** The **nice** value for the process group should be returned.

**PRIO_PROCESS**
The **nice** value for the process should be returned.

**PRIO_USER** The **nice** value associated with the user ID should be returned.

*who* Specifies a numeric value interpreted relative to the *which* parameter (a process group ID, an OSS process ID, and a user ID, respectively). A 0 (zero) value for the *who* parameter indicates the current process group ID, OSS process ID, or user ID.

**DESCRIPTION**

The **getpriority( )** function obtains the nice value of a process group, process, or user ID.

The **getpriority( )** function returns the highest priority (lowest numerical value) pertaining to any of the specified processes.

**Use From the Guardian Environment**

This function cannot be called from the Guardian environment. When the **getpriority( )** function is called from the Guardian environment, the call fails and **errno** is set to [ENOTOSS].

**RETURN VALUES**

Because **getpriority( )** can legitimately return the value of -1, set the external variable **errno** to 0 (zero) before calling the **getpriority( )** function. If a value of -1 is returned from **getpriority( )**, check **errno** to see whether an error occurred or the value is a legitimate priority.

Upon successful completion, the **getpriority( )** function returns an integer in the range -20 through 19. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **getpriority( )** function sets **errno** to the corresponding value.

[EINVAL] One of the following conditions occurred:

- The value specified by the *which* parameter is invalid.
- The value specified as a process group ID, OSS process ID, or user ID by the *who* parameter is out of range.

[ENOTOSS] The calling process is not an OSS process. The requested operation is not supported from the Guardian environment.

[ESRCH] No process was located using the *which* and *who* parameter values specified.

## RELATED INFORMATION

Functions: **nice(2)**.

## STANDARDS CONFORMANCE

The following is an HP extension to the XPG4 Version 2 specification:

- The function can return the **errno** value [ENOTOSS].

## NAME

**getsid** - Gets the process group ID of the session leader

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <unistd.h>**

**pid_t getsid(**
**pid_t** *pid***);**

## PARAMETERS

*pid* Specifies the OSS process ID of the target process. If the value specified is **pid_t(0)** (an OSS process ID of zero), the function uses the OSS process ID of the calling process.

## DESCRIPTION

The **getsid( )** function returns the process group ID of the process that is the session leader of the process specified by the OSS process ID in the *pid* parameter. Specifying a *pid* of 0 (zero) returns the process group ID of the calling process.

### Use From the Guardian Environment

Calls to **getsid( )** from Guardian processes are not successful because Guardian processes do not have OSS process IDs. Such calls return an **errno** value of [ENOTOSS].

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **getsid( )** function returns the process group ID of the specified process. If the function call fails, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **getsid( )** function sets **errno** to the corresponding value:

[ENOTOSS] The calling process is not an OSS process. The requested operation is not supported from the Guardian environment.

[EPERM] The specified process is not in the same session as the calling process, and the calling process lacks sufficient privilege to read the specified process.

[ESRCH] No process has been found that has an OSS process ID identical to that specified by the *pid* parameter.

## RELATED INFORMATION

Functions: **execl(2)**, **execle(2)**, **execlp(2)**, **execv(2)**, **execve(2)**, **execvp(2)**, **fork(2)**, **setsid(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**.

## STANDARDS CONFORMANCE

The following is an HP extension to the XPG4 Version 2 specification:

- The **errno** value [ENOTOSS] can be returned.

## NAME

**getsockname -** Gets the locally bound name of a socket

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <sys/socket.h>**

**int getsockname(**
 **int** *socket***,**
 **struct sockaddr *** *address***,**
 **socklen_t *** *address_len***);**

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*address* Points to a **sockaddr** structure, where the locally bound address of the specified socket is returned.

If the length of the socket address is greater than the length of the supplied **sockaddr** structure, the address is truncated when stored.

If the socket has not been bound to a local address, the value stored is unspecified.

The length and format of the address depend on the address family of the socket.

For **AF_INET** sockets, a pointer to the address structure **sockaddr_in** can be cast as a **struct sockaddr**. For **AF_INET6** sockets, a pointer to the address structure **sockaddr_in6** can be cast as a **struct sockaddr**. For **AF_UNIX** sockets, a pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*address_len* Points to a **socklen_t** data item, which, on input, specifies the length of the **sockaddr** structure pointed to by the *address* parameter, and, on output, specifies the length of the address returned.

## DESCRIPTION

The **getsockname( )** function retrieves the locally-bound address of the specified socket, stores this address in the **sockaddr** structure pointed to by the *address* parameter, and stores the length of this address in the object pointed to by the *address_len* parameter.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

A process can use the **getpeername( )** function to retrieve the name of a peer socket in a socket connection.

After a successful call to **bind( )**, if the underlying file is unlinked or renamed, the behavior of **getsockname( )** depends on the mode of the socket:

- For **AF_UNIX** Release 1 sockets and **AF_UNIX** Release 2 sockets in compatibility mode, **getsockname( )** fails and **errno** is set to [ENOENT].
- For **AF_UNIX** Release 2 sockets in portability mode, the fully-qualified name of the file is returned.

For more information about **AF_UNIX** Release 2 sockets, portability mode, and compatibility mode, see the *Open System Services Programmer's Guide*.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

**RETURN VALUES**

Upon successful completion, the **getsockname( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occur, the **getsockname( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINVAL] The socket has been shut down.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time might succeed.

[ENOENT] The socket is either an **AF_UNIX** Release 1 datagram socket or an **AF_UNIX** Release 2 datagram socket in compatibility mode and one of these conditions occured:

- The underlying file of the peer socket specified in the *address* parameter was renamed.
- The underlying file of the peer socket specfied in the *address* parameter was unlinked.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified operation is not supported by the protocol used by the socket.

**RELATED INFORMATION**

Functions: **accept(2)**, **bind(2)**, **getpeername(2)**, **socket(2)**.

**STANDARDS CONFORMANCE**

The HP implementation does not return the **errno** value [ENOSR].

The following are HP extensions to the XPG4 specification:

- The **errno** value [ECONNRESET] can be returned when the transport-provider process is unavailable.

## NAME

**getsockopt -** Gets socket options

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zinetsrl**
H-series and J-series OSS processes: **/G/system/zdll***nnn***/zinetdll**

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <sys/socket.h>**
[**#include <netinet/in.h>**] Required for IP protocol level
[**#include <netinet/tcp.h>**] Required for TCP protocol level

**int getsockopt(**
  **int** *socket***,**
  **int** *level***,**
  **int** *option_name***,**
  **void ****option_value***,**
  **socklen_t ****option_len***);**

## PARAMETERS

*socket* Specifies the file descriptor for the socket.

*level* Specifies the protocol level at which the option resides. The following values can be specified for the *level* parameter in an OSS application program:

**IPPROTO_IPV6**
Return IP protocol-level options defined for an Internet Protocol version 6 (IPv6) socket

**IPPROTO_IP** Return IP protocol-level options defined for an Internet Protocol version 4 (IPv4) socket

**IPPROTO_TCP**
Return TCP protocol-level options defined for a socket

**SOL_SOCKET**
Return socket-level protocol options defined for a socket

To retrieve options at other levels, supply the appropriate protocol number for the protocol controlling the option. Supported protocol numbers are listed in **/etc/protocols**.

*option_name* Specifies a single option to be retrieved.

The **getsockopt( )** function retrieves information about the following **IPPROTO_IPV6** (IP protocol-level IPv6) options:

**IPV6_MULTICAST_IF**
Indicates the interface (subnet) used for outbound multicast UDP datagrams. The interface value is an **unsigned int**.

**IPV6_MULTICAST_HOPS**
Indicates the hop limit for outbound multicast UDP datagrams. The limit is an **int** that is either:

- Between 0 and 255 to indicate the maximum number of hops allowed.

- -1 to indicate that the default value should be used.

**IPV6_MULTICAST_LOOP**
Indicates that the host belongs to the multicast group that it is sending to, and a copy of the datagram should be sent by loopback to the originating host.

**IPV6_UNICAST_HOPS**
Indicates the hop limit for outbound unicast UDP datagrams. The limit is an **int** that is either:

- Between 0 and 255 to indicate the maximum number of hops allowed.
- -1 to indicate that the default value should be used.

**IPV6_V6ONLY**
Indicates that **AF_INET6** sockets are restricted to IPv6-only communication.

The **getsockopt( )** function retrieves information about the following **IPPROTO_IP** (IP protocol-level IPv4) options:

**IP_OPTIONS** Indicates that the value of the **IP_OPTIONS** flag should be returned. The **IP_OPTIONS** flag indicates that options specified in a **setsockopt( )** function call are set for each outgoing packet, in conformance with RFC 791.

**IP_MULTICAST_IF**
Indicates the interface (subnet) used for outbound multicast UDP datagrams. The interface value is an **unsigned int**.

**IP_MULTICAST_TTL**
Indicates the hop limit for outbound multicast UDP datagrams. The limit is an **int** that is either:

- Between 0 and 255 to indicate the maximum number of hops allowed.
- -1 to indicate that the default value should be used.

**IP_MULTICAST_LOOP**
Indicates that the host belongs to the multicast group that it is sending to, and a copy of the datagram should be sent by loopback to the originating host.

The **getsockopt( )** function retrieves information about the following **SOL_SOCKET** (socket protocol-level) options:

**SO_ACCEPTCONN**
Reports whether socket listening is enabled. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**SO_BROADCAST**
Reports whether transmission of broadcast messages is supported. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**SO_DEBUG** Reports whether debugging information is being recorded. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**SO_DONTROUTE**
Reports whether outgoing messages should bypass the standard routing facilities and be directed to the appropriate network interface, according to the destination address. (This option is for debugging purposes only and is not recommended.) This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**SO_ERROR** Reports information about error status and clears it. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**SO_KEEPALIVE**
Reports whether connections are kept active with periodic transmission of messages. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**SO_LINGER** Reports whether the system attempts to deliver data after the **close( )** function is called if unsent data is queued.

The **SO_LINGER** option is always enabled for **AF_INET** or **AF_INET6** sockets and is not implemented for **AF_UNIX** sockets.

This option returns a **linger** structure value in the buffer pointed to by the *option_value* parameter.

**SO_OOBINLINE**
Reports whether the socket leaves received out-of-band data (data marked urgent) queued with other data (in line) for protocols that support out-of-band data. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**SO_RCVBUF** Reports the receive buffer size in bytes. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**SO_REUSEADDR**
Reports whether the rules used in validating addresses supplied by a **bind( )** function call should allow reuse of local addresses. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**SO_REUSEPORT**
Reports whether the rules used in validating ports supplied by a **bind( )** function call should allow reuse of local ports. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

This option is valid only for UDP ports.

**SO_SNDBUF** Reports the send buffer size in bytes. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**SO_TYPE** Reports the socket type in a form that can be tested against a symbolic value such as **SOCK_DGRAM** or **SOCK_STREAM**. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

The **getsockopt(\|)** function retrieves information about the following **IPPROTO_TCP** (TCP protocol-level) options:

**TCP_MAXRXMT**
Reports the maximum retransmission timeout value in multiples of 500 milliseconds. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**TCP_MINRXMT**
Reports the minimum retransmission timeout value in multiples of 500 milliseconds. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**TCP_NODELAY**
Reports whether data packets are buffered before transmission. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**TCP_RXMTCNT**
Reports the maximum retransmission count. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**TCP_SACKENA**
Reports whether TCP selective acknowledgments are enabled. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

**TCP_TOTRXMTVAL**
Reports the total maximum retransmission duration in multiples of 500 milliseconds. This option returns an **int** value in the buffer pointed to by the *option_value* parameter.

Options at other protocol levels vary in format and name.

*option_value* Points to the buffer to receive the option value. The data type of the value returned for each option is indicated in the description of *option_name*.

If the length of the value returned for an option is greater than the length of the supplied *option_value* buffer, the value is truncated when stored.

*option_len* Points to a **socklen_t** data item, which, on input, specifies the length of the supplied buffer pointed to by the *option_value* parameter, and, on output, specifies the length of the value returned in the supplied buffer.

## DESCRIPTION

The **getsockopt( )** function allows an application program to query socket options. The calling program specifies the file descriptor, the option, and a place to store the requested information. The operating system gets the socket option information from its internal data structures and passes the requested information back to the calling program.

Upon successful completion, the **getsockopt( )** function returns the value of the specified option in the buffer pointed to by the *option_value* parameter. For options that can be classified as disabled or enabled, a value of 0 (zero) indicates that the option is disabled and a value of 1 indicates that the option is enabled. The socket-level options can be enabled or disabled by the **setsockopt( )** function.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **getsockopt( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **getsockopt( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINVAL] One of the following conditions occurred:

- The specified option is not valid at the specified socket level.
- The socket has been shut down.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time might succeed.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOPROTOOPT]
The specified option is not supported by the protocol used by the socket.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified operation is not supported by the protocol used by the socket.

**RELATED INFORMATION**

Functions: **bind(2)**, **close(2)**, **endprotoent(3)**, **getprotobynumber(3)**, **getprotoent(3)**, **setprotoent(3)**, **setsockopt(2)**, **socket(2)**, **socketpair(2)**.

**STANDARDS CONFORMANCE**

The HP implementation does not:

- Implement the **SO_LINGER** option for **AF_UNIX** sockets.
- Return the **errno** value [ENOSR].

The following are HP extensions to the XPG4 specification:

- The **errno** value [ECONNRESET] can be returned when the transport provider process is unavailable.
- The **SO_REUSEPORT** option is supported.

## NAME

**gettimeofday** - Gets date and time

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

## SYNOPSIS

```
#include <sys/time.h>

int gettimeofday(
        struct timeval *tp,
        struct timezone *tzp);
```

## PARAMETERS

*tp* Points to a **timeval** structure, defined in the **sys/time.h** file. The **timeval** structure contains the current time when the call is completed. If a null pointer is specified, the current time is not returned. If an invalid writable address is specified, a SIGSEGV signal is generated.

*tzp* Points to a **timezone** structure, defined in the **sys/time.h** file. The **timezone** structure contains the current time zone when the call is completed. If a null pointer is specified, the current time zone is not returned. If an invalid writable address is specified, a SIGSEGV signal is generated.

## DESCRIPTION

The **gettimeofday( )** function gets the system values for the current time and time zone. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970.

The **timeval** structure contains the following fields:

**tv_sec** The number of whole seconds of elapsed time.

**tv_usec** The number of additional microseconds of elapsed time.

The **timezone** structure contains the following fields:

**tz_minuteswest**
The local time zone, measured in minutes of time westward from Coordinated Universal Time (Greenwich, England).

**tz_dsttime** A value that indicates which daylight-saving time correction applies locally during the appropriate part of the year.

This value is always set to 0 (zero) upon successful completion, because the information required to return an accurate nonzero value is not available.

### Use From the Guardian Environment

This function can be used in the Guardian environment.

## NOTES

The **gettimeofday( )** function is supported for compatibility with BSD programs. This function provides a process-local time-zone parameter in addition to the systemwide time and date.

**RETURN VALUES**

Upon completion, the value 0 (zero) is returned.

**RELATED INFORMATION**

Functions: **ctime(3)**, **strftime(3)**.

Commands: **date(1)**.

**STANDARDS CONFORMANCE**

In the HP implementation:

- The *tzp* parameter is not a type **void** data item.
- The **tz_dsttime** field is always set to 0 (zero).

The following are HP extensions to the XPG4 Version 2 specification:

- The behavior when the *tzp* parameter is a null pointer is specified.

## NAME

**getuid -** Gets the the real user ID of the current process

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsecsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zsecdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/ysecdll**

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>

uid_t getuid(void);
```

## DESCRIPTION

The **getuid( )** function returns the real user ID of the current process.

## NOTES

A process's real user ID is not always numerically equal to its operating system creator access ID (CAID).

## RETURN VALUES

The **getuid( )** function returns the requested user ID. It is always successful.

When the authentication information for a process is uninitialized, the "uninitialized" user ID (hexadecimal 80000000) is returned.

## RELATED INFORMATION

Functions: **geteuid(2)**, **setuid(2)**.

**NAME**

**ioctl** - Controls device files

**LIBRARY**

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

**SYNOPSIS**

**#include <stropts.h>**
**#include <sys/ioctl.h>**

**int ioctl(**
    **int** *filedes***,**
    **int** *request***,**
    **/*** *arg* ***/ . . .);**

The ellipsis (**. . .**) indicates that the function is variable.

**PARAMETERS**

*filedes* Specifies the open file descriptor of the tty device or socket.

*request* Specifies the function to be performed for the tty device or socket.

*arg* A pointer to data to be used by the function or to be provided by the function.

**DESCRIPTION**

The **ioctl( )** function controls the operations of devices. The requests that **ioctl( )** performs on devices are device-specific.

The **ioctl( )** function passes the *request* parameter to the file designated by the open file descriptor *filedes*. If the *filedes* parameter identifies an unsupported device type, the function call fails, and **errno** is set to the value [EINVAL].

Valid values for the *request* parameter for **AF_INET** or **AF_INET6** sockets are:

**FIONREAD** Gets the number of bytes available for reading and stores it at the **int** pointed at by *arg*.

**SIOCATMARK**
Examines whether the socket is at an out-of-band data mark and stores it at the **int** pointed at by *arg*. A nonzero value indicates that the socket is at an out-of-band data mark; a zero value indicates that the socket is not at an out-of-band data mark.

**SIOCGIFNUM**
Gets the number of interfaces that have been configured and stores it at the **int** pointed at by *arg*.

**FIONBIO** Sets the blocking I/O/nonblocking I/O flag for the socket. If the *arg* is zero, the socket is set to blocking I/O; otherwise, the socket is set to nonblocking I/O and the flag is stored at the **int** pointed at by *arg*.

**SIOCADDRT** Adds a route. The data structure pointed to by *arg* is of type **rtentry**.

**SIOCDELRT** Deletes a route. The data structure pointed to by *arg* is of type **rtentry**.

**SIOCSIFADDR**
Sets the interface address. The data structure pointed to by *arg* is of type **ifreq**. Returns the error [EOPNOTSUPP].

**SIOCSIFDSTADDR**
Sets the destination address on a point-to-point interface. The data structure pointed to by *arg* is of type **ifreq**. Returns the error [EOPNOTSUPP].

**SIOCSIFFLAGS**
Sets the interface flags. The data structure pointed to by *arg* is of type **ifreq**. Returns the error [EOPNOTSUPP].

**SIOCSIFBRDADDR**
Sets the destination address on a broadcast interface. The data structure pointed to by *arg* is of type **ifreq**. Returns the error [EOPNOTSUPP].

**SIOCSIFNETMASK**
Sets the network address mask, which specifies the portion of the IP host ID and IP network number that should be masked to define a subnet. The data structure pointed to by *arg* is of type **ifreq**. Returns the error [EOPNOTSUPP].

**SIOCSARP** Sets the ARP protocol address entry in the translation table. The data structure pointed to by *arg* is of type **arpreq**.

**SIOCDARP** Deletes the ARP protocol address entry from the translation table. The data structure pointed to by *arg* is of type **arpreq**.

**SIOCGIFADDR**
Gets the interface address. The data structure pointed to by *arg* is of type **ifreq**. Returns the error [ENXIO].

**SIOCGIFDSTADDR**
Gets the destination address on a point-to-point interface. The data structure pointed to by *arg* is of type **ifreq**.

**SIOCGIFFLAGS**
Gets the interface flags. The data structure pointed to by *arg* is of type **ifreq**.

**SIOCGIFBRDADDR**
Gets the destination address on a broadcast interface. The data structure pointed to by *arg* is of type **ifreq**.

**SIOCGIFCONF**
Gets the interface configuration list. The data structure pointed to by *arg* is of type **ifconf**.

**SIOCGIFNETMASK**
Gets the network address mask, which specifies the portion of the IP host ID and IP network number that should be masked to define a subnet. The data structure pointed to by *arg* is of type **ifreq**.

**SIOCGARP** Gets the ARP protocol address entry from the translation table. The data structure pointed to by *arg* is of type **arpreq**.

The values valid for the *request* parameter for tty devices are:

**TIOCGWINSZ**
Causes the current values for the Telserv window identified by the *filedes* parameter to be stored in the **winsize** structure pointed to by *arg*.

**TIOCSWINSZ**

Causes the values stored in the **winsize** structure pointed to by *arg* to be sent to the Telserv window identified by the *filedes* parameter.

When the *request* parameter specifies either **TIOCGWINSZ** or **TIOCSWINSZ**, the third and only additional parameter is a pointer to:

```
struct winsize {
    unsigned short    ws_row;
    unsigned short    ws_col;
    unsigned short    ws_xpixel;
    unsigned short    ws_ypixel;
};
```

The **winsize** structure contains these fields:

*ws_row* The number of rows, in characters, contained in the window

*ws_col* The number of columns, in characters, contained in the window

*ws_xpixel* The horizontal size, in pixels, of the window (zero if the size is not known or if pixel values are not meaningful)

*ws_ypixel* The vertical size, in pixels, of the window (zero if the size is not known or if pixel values are not meaningful)

If the function is called with a *request* value of **TIOCSWINSZ** and if a value in the **winsize** structure has changed since the last call, a **SIGWINCH** signal is sent to all processes in the foreground group.

### Use From the Guardian Environment

A Guardian process cannot receive the **SIGWINCH** signal.

### Use on Guardian Objects

The *filedes* parameter can specify a terminal file in **/G**.

## NOTES

If your application uses the Cluster I/O Protocols (CIP) subsystem, options for this function might not be supported or might result in behaviors that are different from those described in this reference page. For more information about the Cluster I/O Protocols, see the *Cluster I/O Protocols (CIP) Configuration and Management Manual*.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

If the **ioctl( )** function succeeds, it returns a value different from -1. The value returned depends on the device-control function. If an error occurs, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **ioctl( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not a valid descriptor.

[EFAULT] The optional parameter is to be used as an address, but it points outside the process address space.

[EINTR] A signal was caught during execution of the function.

[EINVAL] The function was called for a device other than a terminal.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node (a remote $ZTNT process), but communication with the remote node has been lost.

[ENOTSUP] The *request* parameter specifies an operation that is not supported on the device specified by the *filedes* parameter.

[ENOTTY] The *device* parameter is not associated with a character-special device, or the specified request does not apply to the type of object that descriptor *device* references.

[ENXIO] No such device or address exists.

[EOPNOTSUPP]
Operation not supported on socket. The type of socket (address family or protocol) does not support the requested operation.

[EWRONGID] One of these conditions occurred:

- The process attempted an input or output operation on an input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

**RELATED INFORMATION**

Files: **tty(7)**.

**STANDARDS CONFORMANCE**

The OSS version of this function does not conform to a published standard.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EFAULT], [ENETDOWN], [ENOTSUP], [ENXIO], [EOPNOTSUPP], and [EWRONGID] can be returned.

# Section 4. System Functions (k - m)

This section contains reference pages for Open System Services (OSS) system function calls with names that begin with **k** through **m**. These reference pages reside in the **cat2** directory and are sorted alphabetically by U.S. English conventions in this section.

## NAME

**kill -** Sends a signal to a process or group of processes

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <signal.h>

int kill(
        pid_t pid,
        int signal);
```

## PARAMETERS

*pid* Specifies the process or group of processes to be sent a signal.

*signal* Specifies the signal. If the *signal* parameter has the value 0 (zero, the null signal), error checking is performed but no signal is sent. The null signal can be used to check the validity of the *pid* parameter.

## DESCRIPTION

The **kill( )** function sends the signal specified by the *signal* parameter to the process or group of processes specified by the *pid* parameter.

To send a signal to another process, at least one of the following must be true:

- The real or effective user ID of the sending process must match the real or saved-set-user ID of the receiving process.
- The process is trying to send the **SIGCONT** signal to a process in the same session.
- The calling process has appropriate privileges.

Processes can send signals to themselves.

### Use on Guardian Objects

The **kill( )** function cannot send signals to Guardian processes.

### Use From the Guardian Environment

If called from a Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

### Specifying the Target Process

The **kill( )** function allows the calling process to send a signal to a specific group of processes. The *pid* parameter specifies the group according to the following rules:

- If the *pid* parameter is greater than 0 (zero), the signal specified by the *signal* parameter is sent to the process that has an OSS process ID (PID) equal to the value of the *pid* parameter.
- If the *pid* parameter is equal to 0 (zero), the signal specified by the *signal* parameter is sent to all of the processes whose process group ID is equal to the process group ID of the sender and for which the process has permission to send the signal.
- If the *pid* parameter is negative but not equal to -1, the signal specified by the *signal* parameter is sent to all of the processes whose process group ID is equal to the absolute value of the *pid* parameter and for which the process has permission to send the signal.

The POSIX.1 standard leaves unspecified the set of system processes that does not receive a signal when the **kill( )** function is called with *pid* equal to 0, -1, or a negative number less than -1. Applications in the HP implementation should therefore not depend on which system processes receive signals.

### Safeguard Considerations

HP recommends that users not use Safeguard protection on OSS processes. However, if users do use Safeguard protection on OSS processes, the following constraints apply.

For unnamed processes, the Safeguard software applies additional protection if the special UNNAMED process protection record exists. For signals whose default action is to terminate the process, the system checks the access control list associated with the UNNAMED protection record and sends the signal only if the caller has purge authority.

When authority is granted to send a signal to a group of unnamed processes, all group members receive the signal in one step if the sending process has authority to send signals to all the processes in the group. Otherwise, the signal is not sent to any member of the group.

For named processes, the Safeguard software applies additional protection if a protection record exists for the specific process name. For signals whose default action is to terminate the process, the system checks the access control list associated with the protection record and sends the signal to the process only if the caller has purge authority.

When the **kill( )** function attempts to send a signal to a group that contains named processes, the signal is sent to the named members of the group one at a time; that is, not all in one step as with unnamed processes. Processes joining or leaving the group between sending the signal to the first process and sending the signal to the last process can therefore affect the result of the combined operation.

### Guardian Process Deletion Messages

When an OSS process terminates, the system performs most of the Guardian process-termination sequence. One of the steps in that sequence is to check if a Guardian Process Deletion (-101) message needs to be sent. The Guardian MOM, GMOM, and ANCESTOR attributes of the process determine the need to send the message and the recipient of the message.

OSS processes normally have null values for these attributes and therefore do not cause Process Deletion messages to be sent. If an OSS process is created using any of the following functions with nonnull values for MOM, GMOM, and ANCESTOR in its **process_extension_def** structure, a Process Deletion message is sent during termination:

**tdm_fork( )**
**tdm_exec** set of functions
**tdm_spawn** set of functions

If an OSS process uses the Guardian PROCESS_SETINFO_ procedure to set nonnull values for MOM, GMOM, and ANCESTOR, a Process Deletion message is sent during termination.

See the *Guardian Procedure Errors and Messages Manual* for details on the Process Deletion message. See the PROCESS_SETINFO_ procedure in the *Guardian Procedure Calls Reference Manual* for information on setting the MOM field.

### Terminated Processes

The **kill( )** function does not return an error when applied to any process that has terminated but whose process lifetime is not yet exhausted. However, the operation has no effect, because the process is already terminated. The termination wait status of the process is unaffected by the operation.

**RETURN VALUES**

Upon successful completion, the **kill()** function returns the value 0 (zero). Otherwise, the value -1 is returned, **errno** is set to indicate the error, and no signal is sent.

**ERRORS**

If any of the following conditions occurs, the **kill()** function sets **errno** to the corresponding value:

[EINVAL] The value in the *signal* parameter is an invalid or unsupported signal number.

[ENOTOSS] The calling process was not an OSS process. The **kill()** function cannot be used from the Guardian environment.

[EPERM] The process does not have permission to send the signal to any receiving process.

[ESRCH] No process or process group can be found corresponding to that specified by the *pid* parameter.

**RELATED INFORMATION**

Functions: **getpid(2)**, **setpgid(2)**, **setsid(2)**, **sigaction(2)**, **signal(3)**.

**STANDARDS CONFORMANCE**

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- The mechanism that grants a process the appropriate privileges to send signals is described under **DESCRIPTION**.
- The POSIX.1 standard leaves unspecified the set of system processes that does not receive a signal when the **kill()** function is called with *pid* equal to 0 (zero), -1, or a negative number less than -1. Applications in the HP implementation should therefore not depend on which system processes receive signals.
- Additional restrictions are imposed by the Safeguard security mechanism on the sending of signals. See **Safeguard Considerations**.

The following are HP extensions to the XPG4 Version 2 specification:

- Safeguard considerations
- The ability to send signals to named processes
- The error [ENOTOSS]

## NAME

**lchmod** - Changes file-access permissions

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/types.h>** /* optional except for POSIX.1 */
**#include <sys/stat.h>**

**int lchmod(**
**const char ****path***,**
**mode_t** *mode***);**

## PARAMETERS

*path* Specifies the full pathname of the file. If the final component of the *path* parameter refers to a symbolic link, the **lchmod( )** function changes access permissions for the symbolic link instead of the file to which it refers.

*mode* Specifies the bit pattern that determines the access permissions.

## DESCRIPTION

The **lchmod( )** function sets the access permissions of a file specified by the *path* parameter according to the bit pattern specified by the *mode* parameter. The **lchmod( )** function is similar to the **chmod( )** function except when the final component specified by the *path* parameter is a symbolic link. If the final component of the *path* parameter refers to a symbolic link, the **lchmod( )** function changes access permissions for the symbolic link instead of the file to which it refers.

Access control lists (ACLs) are not supported for symbolic links.

To change the file access permissions of a file or directory, the effective user ID of the process must match the super ID or the owner of the file, or its effective user ID or one of its group affiliations must qualify it for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group.

If **lchmod( )** is invoked by a process whose effective user ID does not equal the super ID or file owner, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000, respectively) are cleared.

If **lchmod( )** is invoked to set either or both of the set-user-ID and set-group-ID bits of the file mode (04000 and 02000 respectively), then any file privileges the file might have had are cleared.

See also "Accessing Files in Restricted-Access Filesets."

If the **S_ISVTX** bit is on for a directory, only processes with an effective user ID equal to the user ID of the file's owner or the directory's owner, or a process with appropriate privileges, can remove files from the directory.

A call to the **lchmod( )** function has no effect on the file descriptor for a file that is open at the time of the call. However, new openers of the file will be authorized by using the new access permissions that were specified in the call.

The *mode* parameter is constructed by logically ORing one or more of these symbols, which are defined in the **sys/stat.h** header file:

**S_ISUID** Sets the process's effective user ID to the user ID of the file's owner on execution.

**S_ISGID** Sets the process's effective group ID to the group ID of the file's group on execution.

**S_ISVTX** For a directory, permits modification to the directory only if the effective user ID of the process matches that of the file being accessed.

**S_IRWXU** Permits the file's owner to read, write, and execute the file (or to search the directory).

**S_IRUSR** Permits the file's owner to read the file.

**S_IWUSR** Permits the file's owner to write to the file.

**S_IXUSR** Permits the file's owner to execute the file (or to search the directory).

**S_IRWXG** Permits the file's group to read, write, and execute the file (or to search the directory).

**S_IRGRP** Permits the file's group to read the file.

**S_IWGRP** Permits the file's group to write to the file.

**S_IXGRP** Permits the file's group to execute the file (or to search the directory).

**S_IRWXO** Permits others to read, write, and execute the file (or to search the directory).

**S_IROTH** Permits others to read the file.

**S_IWOTH** Permits others to write to the file.

**S_IXOTH** Permits others to execute the file (or to search the directory).

**S_TRUST** Establishes that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers when the memory segment containing the buffers is not shared. This flag applies only to loadfiles for a TNS/E native process and can be set only by a user with appropriate privileges (the super ID).

**S_TRUSTSHARED**
Establishes that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers regardless of whether the memory segment containing the buffers is shared. This flag applies only to loadfiles for a TNS/E native process and can be set only by a user with appropriate privileges (the super ID).

The **S_ISUID** bit of the file is not changed by the call if the file specified by the *path* parameter resides on a node where the calling process is not logged in.

The **S_ISGID** bit of the file is cleared if all of these conditions are true:

- The named file is a regular file.
- The process does not have appropriate privileges.
- The file's group ID does not match the effective group ID of the process or one of the IDs of the process's group list.

Upon successful completion, the **lchmod( )** function marks the **st_ctime** field of the file for update.

### Access Control Lists (ACLs)

When you execute the **lchmod( )** function, you can change the effective permissions granted by optional entries in the ACL for a file. In particular, using the **lchmod( )** function to remove read, write, and execute permissions from a file owner, owning group, and all others works as expected, because the **lchmod( )** function affects the **class** entry in the ACL, limiting any access that can be granted to additional users or groups through optional ACL entries. To verify the effect, use **getacl** command on the file after the **lchmod( )** function completes and note that all optional (nondefault) ACL entries with nonzero permissions also have the comment **# effective:---**.

To set the permission bits of ACL entries, use the **acl( )** function instead of the **lchmod( )** function.

ACLs are not supported for symbolic links.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted. In a restricted-access fileset:

- The super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is not permitted to invoke this function on files that it does not own unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.
- Processes that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, if the executable file for the process does not have the PRIVSOARFOPEN file privilege, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000 respectively) of the file accessed by this function are cleared.
- Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use on Guardian Objects

Attempting to set the access permissions on a Guardian file (that is, a file in the **/G** file system) fails with **errno** set to [EINVAL].

### Use From the Guardian Environment

The **lchmod( )** function is one of a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **lchmod( )** function returns the value 0 (zero). Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occur, the **lchmod( )** function sets **errno** to the corresponding value:

[EACCES] Search permission is denied for a component of the *path* parameter.

[EFAULT] The *path* parameter points to a location outside of the allocated address space of the process.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EINVAL] One of these conditions exists:

- The value of the *mode* parameter is invalid.
- An attempt was made to set access permissions on a Guardian file (that is, a file in the **/G** file system).

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ELOOP] Too many symbolic links were encountered in translating the *path* parameter.

[ENAMETOOLONG]
One of these names is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

You can call the **pathconf( )** function to obtain the applicable limits.

[ENOENT] One of these conditions exists:

- The named file does not exist, or the specified name is an empty string.
- The *path* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOROOT] One of these conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote node, and communication with the remote name server has been lost.

[ENOTDIR] A component, other than the last part, of the *path* parameter is not a directory.

[ENXIO] The fileset containing the client's current working directory or root directory is not mounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] The effective user ID does not match the user ID of the owner of the file, or the owner does not have appropriate privileges.

[EROFS] The named file resides on a read-only fileset.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Commands: **chmod(1)**, **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **chown(2)**, **fcntl(2)**, **getgroups(2)**, **fchmod(2)**, **fchown(2)**, **lchown(2)**, **mknod(2)**, **open(2)**, **open64(2)**, **read(2)**, **setfilepriv(2)**, **write(2)**.

Miscellaneous topics: **acl(5)**.

## STANDARDS CONFORMANCE

This function is an HP extension to the XPG4 Version 2 specification.

## NAME

**lchown -** Changes the owner and group IDs of a file

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>

int lchown(
        const char *path,
        uid_t owner,
        gid_t group);
```

## PARAMETERS

*path* Specifies the name of the file whose owner ID, group ID, or both are to be changed. If the final component of the *path* parameter names a symbolic link, the **lchown** function changes the ownership of the symbolic link instead of the file or directory to which the symbolic link refers.

*owner* Specifies a numeric value representing the owner ID.

*group* Specifies a numeric value representing the group ID.

## DESCRIPTION

The **lchown( )** function changes the owner and group of a file. The **lchown( )** function is equivalent to the **chown( )** function except when the final component of the *path* parameter refers to a symbolic link. If the final component of the *path* parameter names a symbolic link:

- The **lchown** function changes the ownership of the symbolic link instead of the file or directory to which the symbolic link refers.
- Access control lists (ACLs) are not supported for a symbolic link.

Only a process that has an effective user ID equal to the super ID or to the file owner, or that has an effective user ID or group affiliation qualifying for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group can use the **lchown( )** function to change the group of a file. However, processes that have an effective user ID equal to the file owner can only change the group of a file to a group to which they belong (their effective group or one of their supplementary groups).

If the **lchown( )** function is invoked by a process whose effective user ID does not equal the super ID, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000, respectively) are cleared.

See also "Accessing Files in Restricted-Access Filesets."

A process can change the value of the owner ID of an OSS file only if the effective user ID of the process gives the process appropriate privileges. A process can change the value of the file group ID if the effective user ID of the process matches the owner ID of the file or the process has appropriate privileges. A process without appropriate privileges can change the group ID of a file only to the value of its effective group ID or to a value in its group list. However, if the **lchown( )** function is successfully invoked on a file, the **S_ISGID** and **S_ISUID** bits of the **st_mode** field of the **stat** structure are cleared unless the user has appropriate privileges.

The **_POSIX_CHOWN_RESTRICTED** feature is enforced for all files in the OSS file system. Only processes with appropriate privileges can change owner IDs.

If the *owner* or *group* parameter is specified as -1 cast to the type of **uid_t** or **gid_t**, respectively, the corresponding ID of the file is unchanged. To change only one attribute, specify the other as -1.

Upon successful completion, the **lchown( )** function marks the **st_ctime** field of the file for update.

### Access Control Lists (ACLs)

A user can allow or deny specific individuals and groups access to a file by using an ACL on the file. When using the **lchown( )** function with ACLs, if the new owner and/or group of a file have optional ACL entries corresponding to **user:***uid***:***perm* or **group:***gid***:***perm* in the ACL for a file, those entries remain in the ACL but no longer have any effect because they are superseded by the **user::***perm* or **group::***perm* entries in the ACL.

ACLs are not supported for symbolic links.

For more information about ACLs, see the **acl(5)** reference page.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted. In a restricted-access fileset:

- The super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is not permitted to invoke this function on files that it does not own unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

- Processes that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, if the executable file for the process does not have the PRIVSOARFOPEN file privilege, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000 respectively) of the file accessed by this function are cleared.

- Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use on Guardian Objects

You can use **lchown( )** function on Guardian disk files (that is, disk files in the **/G** file system). Attempts to change the ownership of other types of Guardian files fail and set **errno** to [EINVAL].

For Guardian disk files, Guardian security is used, and any user can pass file ownership to any other user. You must specify value other than -1 for the *owner* parameter (that is, an owner ID must be specified). However, changing the owner ID also changes the group ID to the Guardian group ID of the new owner (that is, bits <16:23> of the new access ID). You cannot use the **lchown( )** function to set the group ID for a Guardian file except as a result of changing the owner ID.

The **_POSIX_CHOWN_RESTRICTED** feature is ignored for files in the Guardian file system (that is, for files in **/G**).

### Use From the Guardian Environment

The **lchown( )** function is one of a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **lchown( )** function returns the value 0 (zero). Otherwise, the value **-**1 is returned, the owner and group of the file remain unchanged, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occur, the **lchown( )** function sets **errno** to the corresponding value:

[EACCES] Search permission is denied on a component of the *path* parameter.

[EFAULT] The *path* parameter is an invalid address.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EINVAL] The *owner* or *group* parameter is out of range.

An attempt was made to change ownership of a Guardian file that is not a disk file.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ELOOP] Too many symbolic links were encountered in translating the *path* parameter.

[ENAMETOOLONG]
One of these names is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

You can call **pathconf( )** function to obtain the applicable limits.

[ENOENT] One of these conditions is true:

- The *path* parameter does not exist.
- The *path* parameter is an empty string.
- The *path* parameter specifies a file in the Guardian file system (in **/G**) but cannot be mapped to a valid Guardian filename.
- The *path* parameter names a symbolic link, but the file to which it refers does not exist.
- The *path* parameter specifies a file on a remote node, but communication with the remote node has been lost.

[ENOROOT] One of these conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node, and communication with the remote name server has been lost.

[ENOTDIR] A component of *path* is not a directory.

[ENOTSUP] The *path* parameter specifies a Guardian file on an SMF logical volume and one of the following conditions exists:

- The local system is running an RVU prior to J06.15 or H06.26.
- The *path* parameter specifies a file in /E and the remote system is running an RVU prior to J06.15 or H06.26.

[ENXIO] The fileset containing the client's current working directory or root directory is not mounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The named file resides on a read-only fileset.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Commands: **chgrp(1)**, **chown(1)**, **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **chown(2)**, **fchmod(2)**, **fchown(2)**, **lchmod(2)**, **setfilepriv(2)**.

Miscellaneous topics: **acl(5)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- A process can change the value of the owner ID of a file only if the effective user ID of the process gives the process appropriate privileges.
- Upon successful completion, the set-user-ID attribute (the **S_ISUID** bit) and the set-group-ID attribute (the **S_ISGID** bit) of the file are always cleared.
- The error [EINVAL] can be detected.

HP extensions to the XPG4 Version 2 specification are:

- To change the file access permissions of a file or directory, the effective user ID of the process must match the super ID or the owner of the file, or the effective user ID or one of the group affiliations for the process must qualify the process for membership in the Safeguard SECURITY-OSS-ADMINISTRATOR group.
- The **errno** values [EFAULT], [EFSBAD], [EIO], [ENOROOT], [ENXIO], and [EOSSNOTRUNNING] can be returned.

## NAME

**link -** Creates an additional directory entry for an existing file on the current fileset

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <unistd.h>**

**int link(**
  **const char ***path1***,**
  **const char ***path2***);**

## PARAMETERS

*path1*
: Points to the pathname of an existing file.

  If any component of the *path1* parameter refers to a symbolic link, the link is traversed and pathname resolution continues.

*path2*
: Points to the pathname for the directory entry to be created.

  If any component of the *path2* parameter refers to a symbolic link, the link is traversed and pathname resolution continues.

  If the final component of the *path2* parameter refers to an existing entity, the call fails and **errno** is set to [EEXIST].

## DESCRIPTION

The **link( )** function creates an additional hard link (directory entry) for an existing file. Both the old and the new link share equal access rights to the underlying file. The **link( )** function atomically creates a new link for the existing file and increments the link count of the file by 1.

The pathnames pointed to by both the *path1* and *path2* parameters must reside on the same fileset. If this is not the case, **errno** is set to [EXDEV].

The *path1* parameter must not name a directory; a hard link to a directory cannot be created. Attempting to create a link to a directory fails. The value -1 is returned and **errno** is set to [EPERM].

Attempting to create a link to **/dev/tty** or **/dev/null** or attempting to create a link in the root directory of an OSS fileset to a file named **lost+found** fails and causes **errno** to be set to [EPERM].

The calling process requires the following:

- Execute (search) permission on the directory containing the existing file.
- Execute (search) and write permission on the directory into which the link is being added.

Upon successful completion, the **link( )** function marks the **st_ctime** field of the file for update and marks the **st_ctime** and **st_mtime** fields of the directory containing the new entry for update.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted by the same file permissions and owner privileges as any other user ID: It has no special privileges unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

Executable files that have the PRIVSOARFOPEN privilege and that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use From the Guardian Environment

The **link( )** function is one of a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

### Use on Guardian Objects

Attempting to create a link to a Guardian file (that is, a file within the **/G** file system) fails and causes **errno** to be set to [EINVAL].

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **link( )** function returns the value 0 (zero). If the **link( )** function fails, the value -1 is returned, no link is created, and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **link( )** function sets **errno** to the corresponding value:

[EACCES] The requested link requires writing in a directory with a mode that denies write permission, or a component of the pathname pointed to by either the *path1* or *path2* parameter denies search permission.

[EEXIST] The link named by the *path2* parameter already exists.

[EFAULT] Either the *path1* or *path2* parameter is an invalid address.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EINVAL] The call attempted to create a link to a Guardian file (that is, a file in **/G** or in any directory within **/G**).

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ELOOP] Too many symbolic links were encountered in translating either the *path1* or *path2* parameter.

[EMLINK] The number of links to the file specified by the *path1* parameter would exceed the maximum permitted. The **pathconf( )** function can be called to obtain the configured limit.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path1* parameter
- The pathname pointed to by the *path2* parameter
- A component of the pathname pointed to by the *path1* parameter
- A component of the pathname pointed to by the *path2* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path1* or *path2* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of the following is true:

- The file specified by the *path1* parameter does not exist.
- The *path1* or *path2* parameter is an empty string.
- The *path1* parameter names a symbolic link, but the file to which it refers does not exist.
- The *path1* or *path2* parameter specifies a file on a remote HP NonStop node but communication with the remote node has been lost.

[ENOROOT] One of the following conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node and communication with the remote name server has been lost.

[ENOSPC] The directory in which the entry for the new link is being placed cannot be extended, because there is no space left on the fileset containing the directory.

[ENOTDIR] A component of either pathname prefix is not a directory.

[ENXIO] The fileset containing the client's current working directory or root directory is not mounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] One of the following conditions occurred:

- The file specified by the *path1* parameter is a directory.
- The call attempted to create a link in the root directory of an OSS fileset to a file called **lost+found**.
- The call attempted to create a link to **/dev/tty** or **/dev/null**.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The requested link requires writing to a directory on a read-only fileset.

[EXDEV] The link specified by the *path2* parameter and the file specified by the *path1* parameter are on different filesets.

## RELATED INFORMATION

Commands: **ln(1)**.

Functions: **unlink(2)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- The **link( )** function is not supported between filesets.
- The **link( )** function is not supported for directories.

The following are HP extensions to the XPG4 Version 2 specification:

- The errors [EFAULT], [EFSBAD], [EINVAL], [EIO], [ENOROOT], [ENOTDIR], [ENXIO], and [EOSSNOTRUNNING] can be returned.

## NAME

**listen -** Listens for socket connections and limits the backlog of incoming connections

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <sys/socket.h>**

**int listen(**
  **int** *socket***,**
  **int** *backlog*
  **);**

## PARAMETERS

*socket* Specifies the file descriptor for the socket.

*backlog* Specifies the maximum number of outstanding connections. The effect of this parameter varies according to the connection type and the stack implementation in use. Refer to the **DESCRIPTION** section of this reference page for more information.

## DESCRIPTION

The **listen( )** function marks a connection-oriented socket as accepting connections, and limits the number of outstanding connections in the socket's queue to the value specified by the *backlog* parameter.

For **AF_INET** sockets using conventional TCP/IP, a *backlog* parameter value of less than or equal to 0 allows the socket to accept the number of connections configured for the TCP-LISTEN-QUE-MIN parameter of the transport process. These values can allow up to 5 connections (the default value for TCP-LISTEN-QUE-MIN).

For **AF_INET** or **AF_INET6** sockets using parallel library TCP/IP or TCP/IPv6, a *backlog* parameter value of less than or equal to 0 is ignored. The maximum number of pending connections is always 5.

If the *backlog* parameter value is less than or equal to 0:

- For **AF_UNIX** Release 1 sockets or for **AF_UNIX** Release 2 sockets in compatibility mode, subsequent calls to the **connect( )** function that specify the path name to which the listening socket is bound fail and *errno* is set to [ECONNREFUSED] unless there is a pending **accept( )** function call on the listening socket.

- For **AF_UNIX** Release 2 sockets in portability mode:

  — Subsequent attempts to issue blocking calls to the **connect( )** function that specify the path name to which the listening socket is bound block the calling process until there is a corresponding **accept( )** call to the socket, at which time the **connect( )** function call succeeds. If there is no corresponding **accept( )** call within 2 minutes, the call to the **connect( )** function fails and *errno* is set to [ETIMEDOUT].

  — Subsequent attempts to issue nonblocking calls to the **connect( )** function that specify the path name to which the listening socket is bound fail with *errno* set to [EWOULDBLOCK], unless there is a pending **accept( )** call to the socket, in which case the **connect( )** function call succeeds.

For more information about **AF_UNIX** Release 2 sockets, portability mode, and compatibility mode, see the *Open System Services Programmer's Guide*.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

## RETURN VALUES

Upon successful completion, the **listen( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **listen( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EDESTADDRREQ]
The socket is not bound to a local address, and the protocol does not support listening on an unbound socket.

[EINVAL] One of the following conditions occurred:

- The socket is already connected.
- The socket has been shut down.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time may succeed.

[ENOMEM] Required memory resources were not available. A retry at a later time may succeed.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The socket specified by the *socket* parameter is not a type that supports the **listen( )** function.

## RELATED INFORMATION

Functions: **accept(2)**, **connect(2)**, **socket(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 specification:

- The **errno** value [ECONNRESET] can be returned when the transport provider process is unavailable.
- The behavior when the *backlog* parameter is 0 (zero) is defined.

## NAME

**lseek** - Sets file offset for read or write operation

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <sys/types.h>** /* optional except for POSIX.1 */
**#include <unistd.h>**

**off_t lseek(**
**int** *filedes***,**
**off_t** *offset***,**
**int** *whence***);**

## PARAMETERS

*filedes*
Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function, or the thread-aware **accept( )**, **dup2( )**, or **fcntl( )** function.

*offset*
Specifies a value, in bytes, that is used with the *whence* parameter to set the file pointer. A negative value causes seeking in the reverse direction.

*whence*
Specifies how to interpret the *offset* parameter in setting the file pointer associated with the *filedes* parameter. Values for the *whence* parameter are:

**SEEK_CUR** Sets the file pointer to its current location plus the value of the *offset* parameter.

**SEEK_END** Sets the file pointer to the size of the file plus the value of the *offset* parameter.

**SEEK_SET** Sets the file pointer to the value of the *offset* parameter.

## DESCRIPTION

The **lseek( )** function sets the file offset for the open file specified by the *filedes* parameter. The *whence* parameter determines how the offset is to be interpreted.

The **lseek( )** function allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reading of data in the gap returns bytes with the value 0 (zero) until data is actually written into the gap.

The **lseek( )** function does not, by itself, extend the size of the file.

### Use From a Threaded Application

This function serializes file operations on an open file. If a thread calls **lseek( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

## NOTES

To use the **lseek( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_lseekz(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the resulting pointer location, measured in bytes from the beginning of the file, is returned. For First-in, First-out (FIFO) files, pipes, and character special files, the value 0 (zero) is returned. For character special files, **errno** is not set.

If the **lseek( )** function fails, the file offset remains unchanged, the value -1 cast to the type **off_t** is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the file offset remains unchanged, and the **lseek( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not an open file descriptor.

[EINVAL] One of these conditions exists:

- The *whence* parameter is an invalid value, or the resulting file offset would be an invalid value (that is, a value less than 0 [zero]).
- The *filedes* parameter refers to a file (other than a pipe, FIFO, or directory) on which seeking cannot be performed.

[EISDIR] The *filedes* parameter refers to an OSS directory.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[EOVERFLOW]
The application was compiled in a regular compilation environment or was compiled using the **#define _LARGEFILE64_SOURCE 1** feature test macro (or an equivalent compiler command option), and the application attempted to set the pointer location at a position between 2 gigabytes minus 1 byte and the maximum file offset established when the file was opened.

[ESPIPE] The *filedes* parameter refers to a pipe, FIFO, or socket.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and the backup process took over.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Functions: **fcntl(2)**, **fseek(3)**, **lseek64(2)**, **open(2)**, **open64(2)**, **read(2)**, **spt_lseekz(2)**, **write(2)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- If the **lseek( )** function is called for a pipe or FIFO, the **errno** value [ESPIPE] is returned.
- If the **lseek( )** function is called for a character special file, no **errno** value is returned.
- If the **lseek( )** function is called for any other device on which seeking cannot be performed, the operation fails, and **errno** is set to [EINVAL].

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EINVAL], [EISDIR], [EISGUARDIAN], and [EWRONGID] can be returned.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**lseek64 -** Sets file offset for read or write operation

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>

off64_t lseek64(
        int filedes,
        off64_t offset,
        int whence);
```

## PARAMETERS

*filedes*
: Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

  When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function, or the thread-aware **accept( )**, **dup2( )**, or **fcntl( )** function.

*offset*
: Specifies a value, in bytes, that is used with the *whence* parameter to set the file pointer. A negative value causes seeking in the reverse direction.

*whence*
: Specifies how to interpret the *offset* parameter in setting the file pointer associated with the *filedes* parameter. Values for the *whence* parameter are:

  **SEEK_CUR** Sets the file pointer to its current location plus the value of the *offset* parameter.

  **SEEK_END** Sets the file pointer to the size of the file plus the value of the *offset* parameter.

  **SEEK_SET** Sets the file pointer to the value of the *offset* parameter.

## DESCRIPTION

The **lseek64( )** function is similar to the **lseek( )** function except that, in addition to supporting smaller files, the **lseek64( )** function supports OSS files larger than approximately 2 gigabytes.

An application can explicitly call this function you compile the application using the **#define _LARGEFILE64_SOURCE 1** feature test macro or an equivalent compiler command option.

An application call to **lseek( )** is automatically mapped to this function when you compile the application using the **#define _FILE_OFFSET_BITS 64** feature test macro or an equivalent compiler command option.

The **lseek64( )** function sets the file offset for the open file specified by the *filedes* parameter. The *whence* parameter determines how the offset is to be interpreted.

The **lseek64( )** function allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reading of data in the gap returns bytes with the value 0 (zero) until data is actually written into the gap.

The **lseek64( )** function does not, by itself, extend the size of the file.

### Use From a Threaded Application

This function serializes file operations on an open file. If a thread calls **lseek64( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

## NOTES

To use the **lseek64( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_lseek64z(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the resulting pointer location, measured in bytes from the beginning of the file, is returned. For First-in, First-out (FIFO) files, pipes, and character special files, the value 0 (zero) is returned. For character special files, **errno** is not set.

If the **lseek64( )** function fails, the file offset remains unchanged, the value -1 cast to the type **off_t** is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the file offset remains unchanged, and the **lseek64( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not an open file descriptor.

[EINVAL] One of these conditions exists:

- The *whence* parameter is an invalid value, or the resulting file offset would be an invalid value (that is, a value less than 0 [zero]).
- The *filedes* parameter refers to a file (other than a pipe, FIFO, or directory) on which seeking cannot be performed.

[EISDIR] The *filedes* parameter refers to an OSS directory.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[EOVERFLOW]
The application attempted to set the file offset beyond the maximum file offset supported for the file.

[ESPIPE] The *filedes* parameter refers to a pipe, FIFO, or socket.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and the backup process took over.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

**RELATED INFORMATION**

Functions: **fcntl(2)**, **fseek(3)**, **open(2)**, **open64(2)**, **read(2)**, **spt_lseek64z(2)**, **write(2)**.

**STANDARDS CONFORMANCE**

This function is an HP extension to the XPG4 Version 2 specification.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**lstat -** Provides information about a symbolic link or any file

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/stat.h>

int lstat(
        const char *path,
        struct stat *buffer);
```

## PARAMETERS

*path* Points to the pathname of a file. If used for a symbolic link, the *path* parameter points to the pathname of the symbolic link identifying the file.

*buffer* Points to a **stat** structure, into which information is placed about the file.

## DESCRIPTION

The **lstat( )** function obtains information about the symbolic link whose name is pointed to by the *path* parameter or about any file pointed to by the *path* parameter.

The **lstat( )** function is like the **stat( )** or **fstat( )** function, except that **lstat( )** returns information about the link, while the **stat( )** and **fstat( )** functions return information about the file that the link refers to.

Read, write, or execute permission for the named file is not required, but all directories listed in the pathname leading to the file must be searchable.

The file information is written to the area specified by the *buffer* parameter, which is a pointer to a **stat** structure. For J06.11 and later J-series RVUs and H06.22 and later H-series RVUs, the **stat** structure uses this definition from the **sys/stat.h** header file:

```
struct stat {
        dev_t    st_dev;
        ino_t    st_ino;
        mode_t   st_mode;
        nlink_t  st_nlink;
        unsigned int      st_acl:1;
        unsigned int      __filler_1:7;
        unsigned int      st_fileprivs:8; /* File privileges */
        uid_t    st_uid;
        gid_t    st_gid;
#if _FILE_OFFSET_BITS != 64 || _TANDEM_ARCH_ == 0
        mode_t   st_basemode; /* Permissions with original group perms */
#endif
        dev_t    st_rdev;
        off_t    st_size;
        time_t   st_atime;
        time_t   st_mtime;
        time_t   st_ctime;
#if _FILE_OFFSET_BITS == 64 && _TANDEM_ARCH_ != 0
        mode_t   st_basemode; /* Permissions with original group perms */
#endif
```

```
 int64_t  st_reserved[3];
};
```

For J06.10 and earlier J-series RVUs and H06.21 and earlier H-series RVUs, the **stat** structure uses this definition from the **sys/stat.h** header file:

```
struct  stat {
            dev_t      st_dev;
            ino_t      st_ino;
            mode_t     st_mode;
            nlink_t    st_nlink;
            unsigned int          st_acl:1;
            unsigned int          __filler_1:15;
            uid_t      st_uid;
            gid_t      st_gid;
#if _FILE_OFFSET_BITS != 64 || _TANDEM_ARCH_ == 0
            mode_t     st_basemode; /* Permissions with original group perms */
#endif
            dev_t      st_rdev;
            off_t      st_size;
            time_t     st_atime;
            time_t     st_mtime;
            time_t     st_ctime;
#if _FILE_OFFSET_BITS == 64 && _TANDEM_ARCH_ != 0
            mode_t     st_basemode; /* Permissions with original group perms */
#endif
            int64_t    st_reserved[3];
};
```

For symbolic links to local OSS objects, the **st_mode** and **st_size** fields are valid and the other fields in the structure are undefined. For symbolic links that resolve to files in **/E**, the **st_dev**, **st_ino**, **st_atime**, **st_mtime**, and **st_ctime** fields are defined as described in this reference page.

For files other than a symbolic link, the **lstat( )** function sets the **st_size** field of the **stat** structure to the length in characters of the absolute pathname resulting from the resolution of the name pointed to by *path*. For a symbolic link, the **lstat( )** function sets the **st_size** field of the **stat** structure to the length in characters of the link name used as the pathname pointed to by *path* (not including the null terminator).

The **lstat( )** function also sets the **st_mode** field to indicate the file type.

The **lstat( )** function updates any time-related fields associated with the file before writing into the **stat** structure, unless it is a read-only fileset. Time-related fields are not updated for read-only OSS filesets.

The fields in the **stat** structure have the following meanings and content:

**st_dev** OSS device identifier for a fileset.

Values for local OSS objects are listed in the following table. Values for local Guardian objects are described in **Use on Guardian Objects**, and values for remote Guardian or OSS objects are described in **Use on Remote Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | ID of device containing directory entry |
| Directory | ID of device containing directory |
| FIFO | ID of special fileset for pipes |
| **AF_UNIX** socket | ID of device containing the fileset in which the socket file was created |
| **/dev/null** | ID of device containing directory entry |
| **/dev/tty** | ID of device containing directory entry |

**st_ino** File serial number (inode number). The file serial number and OSS device identifier uniquely identify a regular OSS file within an OSS fileset.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | File serial number (unique) |
| Directory | File serial number (unique) |
| FIFO | File serial number (unique) |
| **AF_UNIX** socket | File serial number of the socket file (unique) |
| **/dev/null** | File serial number (unique) |
| **/dev/tty** | File serial number (unique) |

The **st_ino** value for all node entries in **/E** (including the entry for the logical link from the local node name to the root fileset on the local node) is the value for the root fileset on the corresponding node. If normal conventions are followed, this value is always 0 (zero), so entries in **/E** appear to be nonunique. Values for objects on remote nodes are unique only among the values for objects within the same fileset on that node.

**st_mode** File mode. The following bits are ORed into the **st_mode** field:

**S_IFMT** File type. This field can contain one of the following values:

**S_IFCHR** Character special file.

**S_IFDIR** Directory.

**S_IFIFO** FIFO.

**S_IFREG** Regular file.

**S_IFSOCK** Socket.

For an **AF_UNIX** socket, the user permissions from the inode for the socket are returned for the permission bits. The access flags are also returned from the inode.

| | |
|---|---|
| **S_IRWXG** | Permissions for the owning group, or if the **st_acl** flag is set, permissions for the the **class** ACL entry. |
| **S_IRWXO** | Other class |
| **S_IRWXU** | Owner class |
| **S_ISGID** | Set group ID on execution |
| **S_ISUID** | Set user ID on execution |
| **S_ISVTX** | Sticky bit; used only for directories (not ORed for files in **/G**, the Guardian file system) |

Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

**st_nlink** Number of links.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Number of links to the file |
| Directory | Number of links to the directory |
| FIFO | Number of links to the file |
| **AF_UNIX** socket | Number of links to the socket file |
| **/dev/null** | Number of links to the file |
| **/dev/tty** | Number of links to the file |

**st_acl** If set to 1, indicates that the file has optional access control list (ACL) entries. For compatibility with HP-UX, the member name **st_aclv** is provided as alias for **st_acl**. For more information about ACLs, see the **acl(5)** reference page.

**st_fileprivs** File privileges. For information about file privileges see the **setfilepriv(2)** reference page.

**st_uid** User ID.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | User ID of the file owner |
| Directory | User ID of the file owner |
| FIFO | User ID of the file owner |
| **AF_UNIX** socket | User ID of the creator of the socket file |
| **/dev/null** | User ID of the super ID |
| **/dev/tty** | User ID of the super ID |

**st_gid** Group ID.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Group ID of the file group |
| Directory | Group ID of the file group |
| FIFO | Group ID of the file group |
| **AF_UNIX** socket | Group ID of the creater of the socket file |
| **/dev/null** | Group ID of the super ID |
| **/dev/tty** | Group ID of the super ID |

**st_basemode** If the **st_acl** flag is set, contains the permissions for the file owner, owning group, and others. If the **st_acl** flag is not set, **st_basemode** is 0 (zero).

**st_rdev** Remote device ID.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Undefined |
| Directory | Undefined |
| FIFO | Undefined |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | Undefined |
| **/dev/tty** | ID of the device |

**st_size** File size.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Size of the file in bytes |
| Directory | 4096 |
| FIFO | 0 (zero) |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | 0 (zero) |
| **/dev/tty** | 0 (zero) |

**st_atime** Access time.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Time of the last access |
| Directory | Time of the last access |
| FIFO | Time of the last access |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_mtime** Modification time.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Time of the last data modification |
| Directory | Time of the last modification |
| FIFO | Time of the last data modification |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_ctime** Status change time.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Time of the last file status change |
| Directory | Time of the last file status change |
| FIFO | Time of the last file status change |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

### Use on Guardian Objects

You can use the **lstat( )** function like the **stat( )** or **fstat( )** function on files in **/G**, but you cannot create symbolic links in **/G**.

The **st_dev** and **st_ino** fields of the **stat** structure do not uniquely identify Guardian files (files in **/G**).

The **st_dev** field is unique for **/G**, for each disk volume, and for each Telserv process (or other process of subdevice type 30), because each of these is a separate fileset.

The **S_ISGUARDIANOBJECT** macro can indicate whether an object is a Guardian object when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is a Guardian object and **FALSE** otherwise.

The **st_ino** field is a nonunique encoding of the Guardian filename.

The **st_rdev** field contains a minor device number for each **pty***n* entry in **/G/ztnt/**, representing each Telserv process subdevice.

The **st_size** field of an EDIT file (file code 101) is the actual (physical) end of file, not the number of bytes in the file. For directories, **st_size** is set to 4096.

When an OSS function is called for a Guardian EDIT file, the **st_mtime** field is set to the last modification time. The **st_atime** field indicates the last time the file was opened, and the **st_ctime** field is set equal to **st_mtime**. No other time-related fields are updated by OSS functions.

The **st_ctime** and **st_atime** fields for Guardian regular disk files (except for EDIT files) are updated by OSS function calls, not by Guardian procedure calls.

The time fields for **/G**, **/G/***vol*, and **/G/***vol*/*subvol* always contain the current time.

When the *path* parameter points to the name of a Guardian process that is not a process of subtype 30, the **lstat( )** function call fails. The value -1 is returned and **errno** is set to [ENOENT].

The **lstat( )** function always returns access modes of "d---------" when the *path* parameter points to a Guardian subvolume that has a reserved name beginning with ZYQ. The other access modes reported for files in **/G** vary according to the file type.

The following table shows the mapping between Guardian files and their corresponding file types described in the **st_mode** field.

| Example in /G | Guardian File Type | st_mode File Type | Permissions |
|---|---|---|---|
| **/G** | N/A | Directory | r-xr-xr-x |
| *vol* | Disk volume | Directory | rwxrwxrwx |
| *vol*/*subvol* | Subvolume | Directory | rwxrwxrwx |
| *vol*/*subvol*/*fileid* | Disk file | Regular file | See following text |
| *vol*/**#123** | Temporary disk file | Regular file | See following text |
| **ztnt** | Subtype 30 process | Directory | --x--x--x |
| **ztnt/#pty0001** | Subtype 30 process with qualifier | Character special | rw-rw-rw- |
| **vol1/zyq00001** | Subvolume | Directory | --------- |

A Guardian file classified as a directory is always owned by the super ID.

Guardian permissions are mapped as follows:

- Guardian network or any user permission is mapped to OSS other permission.
- Guardian community or group user permission is mapped to OSS group permission.
- Guardian user or owner permission is mapped to OSS owner permission.
- Guardian super ID permission is mapped to OSS super ID permission.
- Guardian read permission is mapped to OSS read permission.
- Guardian write permission is mapped to OSS write permission.
- Guardian execute permission is mapped to OSS execute permission.
- Guardian purge permission is ignored.

Users are not allowed read access to Guardian processes.

OSS file permissions are divided into three groups (owner, group, and other) of three permission bits each (read, write, and execute). The OSS permission bits do not distinguish between remote and local users as Guardian security does; local and remote users are treated alike.

### Use on Remote Objects

The content of the **st_dev** field of the **stat** structure is unique for each node in **/E** because each node is a separate fileset. Values for directories within **/E** are the same as described for objects on the local HP NonStop node.

The **S_ISEXPANDOBJECT** macro can indicate whether an object in the **/E** directory is on a remote HP NonStop node when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is on a remote HP NonStop node and **FALSE** otherwise.

### Use From the Guardian Environment

A Guardian process can use thee **lstat( )** function when you use the **#define _XOPEN_SOURCE_EXTENDED 1** feature test macro or an equivalent compiler command option to compile the process.

The **lstat( )** function belongs to a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. You cannot close these file numbers by calling the Guardian FILE_CLOSE_ procedure.

- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

For J06.08 and earlier J-series RVUs, H06.19 and earlier H-series RVUs, or G-series RVUs, the OSS Network File System (NFS) cannot access OSS objects that have OSS ACLs that contain optional ACL entries.

For J06.09 and later J-series RVUs and H06.20 and later H-series RVUs, access by the OSS Network File System (NFS) to OSS objects that have OSS ACLs that contain optional ACL entries can be allowed, depending upon the NFSPERMMAP attribute value for the fileset that contains the object. For more information about NFS and ACLs, see the **acl(5)** reference page.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **lstat( )** function sets **errno** to the corresponding value:

[EACCES] Search permission is denied for a component of the pathname pointed to by the *path* parameter.

[EFAULT] Either the *buffer* parameter or the *path* parameter points to a location outside of the allocated address space of the process.

[EFSBAD] The program attempted an operation involving a fileset with a corrupted fileset catalog.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[ENAMETOOLONG]
One of these names is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

You can call the **pathconf( )** function to obtain the applicable limits.

[ENOENT] One of the following conditions exists:

- The file specified by the *path* parameter does not exist.

- *path* points to an empty string.
- The specified pathname cannot be mapped to a valid Guardian filename.
- The specified pathname points to the name of a Guardian process that is not of subtype 30.
- The *path* parameter specifies a file on a remote HP NonStop node but communication with the remote node has been lost.

[ENOROOT] The program attempted an operation while the root fileset was unavailable.

[ENOTDIR] A component of the pathname specified by the *path* parameter is not a directory.

[ENOTSUP] The **path** parameter refers to a file on a logical disk volume administered through the Storage Management Foundation (SMF).

[ENXIO] An invalid device or address was specified during an input or output operation on a special file. One of the following events occurred:

- A device was specified that does not exist, or a request was made beyond the limits of the device.
- The fileset containing the requestor's current working directory or root directory is not mounted. This error can occur after failure and restart of an OSS name server process until the fileset has been repaired and remounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EOVERFLOW]
The file size (in bytes) or the file inode number (serial number) cannot be represented correctly in the structure pointed to by the *buffer* parameter.

## RELATED INFORMATION

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **chown(2)**, **lstat64(2)**, **fstat(2)**, **open(2)**, **pipe(2)**, **readlink(2)**, **setfilepriv(2)**, **stat(2)**, **symlink(2)**, **utime(2)**.

Miscellaneous Topics: **acl(5)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EFSBAD], [ENOROOT], [ENOTSUP], [ENXIO], and [EOSSNOTRUNNING] can be returned by the **lstat( )** function.

## NAME

**lstat64 -** Provides information about a symbolic link or any file

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/stat.h>**

**int lstat64(**
  **const char ****path*,
  **struct stat64 ****buffer***);**

## PARAMETERS

*path* Points to the pathname of a file. If used for a symbolic link, the *path* parameter points to the pathname of the symbolic link identifying the file.

*buffer* Points to a **stat64** structure, into which information is placed about the file.

## DESCRIPTION

The **lstat64( )** function is similar to the **lstat( )** function except that, in addition to supporting smaller files, the **lstat64( )** function supports OSS files larger than approximately 2 gigabytes.

An application can explicitly call this function when you compile the application using the **#define _LARGEFILE64_SOURCE 1** feature test macro or an equivalent compiler command option.

An application call to **lstat( )** is automatically mapped to this function when you compile the application using the **#define _FILE_OFFSET_BITS 64** feature test macro or an equivalent compiler command option.

The **lstat64( )** function obtains information about the symbolic link whose name is pointed to by the *path* parameter or about any file pointed to by the *path* parameter.

The **lstat64( )** function is like the **stat64( )** or **fstat64( )** function, except that **lstat64( )** returns information about the link, while the **stat64( )** and **fstat64( )** functions return information about the file that the link refers to.

Read, write, or execute permission for the named file is not required, but all directories listed in the pathname leading to the file must be searchable.

The file information is written to the area specified by the *buffer* parameter, which is a pointer to a **stat64** structure. For J06.11 and later J-series RVUs and H06.22 and later H-series RVUs, the **stat64** structure uses this definition from the **sys/stat.h** header file:

```
struct  stat64 {
        dev_t      st_dev;
        ino64_t    st_ino;
        mode_t     st_mode;
        nlink_t    st_nlink;
        unsigned int      st_acl:1;
        unsigned int      __filler_1:7;
        unsigned int      st_fileprivs:8; /* File privileges */
        uid_t      st_uid;
        gid_t      st_gid;
        dev_t      st_rdev;
        off64_t    st_size;
        time_t     st_atime;
        time_t     st_mtime;
        time_t     st_ctime;
        mode_t     st_basemode; /* Permissions with original group perms */
        int64_t    reserved[3];
};
```

For J06.10 and earlier J-series RVUs and H06.21 and earlier H-series RVUs, the **stat64** structure uses this definition from the **sys/stat.h** header file:

```
struct  stat64 {
        dev_t      st_dev;
        ino64_t    st_ino;
        mode_t     st_mode;
        nlink_t    st_nlink;
        unsigned int      st_acl:1;
        unsigned int      __filler_1:15;
        uid_t      st_uid;
        gid_t      st_gid;
        dev_t      st_rdev;
        off64_t    st_size;
        time_t     st_atime;
        time_t     st_mtime;
        time_t     st_ctime;
        mode_t     st_basemode; /* Permissions with original group perms */
        int64_t    reserved[3];
};
```

For symbolic links to local OSS objects, the **st_mode** and **st_size** fields are valid and the other fields in the structure are undefined. For symbolic links that resolve to files in **/E**, the **st_dev**, **st_ino**, **st_atime**, **st_mtime**, and **st_ctime** fields are defined as described in this reference page.

For files other than a symbolic link, the **lstat64( )** function sets the **st_size** field of the **stat64** structure to the length in characters of the absolute pathname resulting from the resolution of the name pointed to by *path*. For a symbolic link, the **lstat64( )** function sets the **st_size** field of the **stat64** structure to the length in characters of the link name used as the pathname pointed to by *path* (not including the null terminator).

The **lstat64( )** function also sets the **st_mode** field to indicate the file type.

The **lstat64( )** function updates any time-related fields associated with the file before writing into the **stat64** structure, unless it is a read-only fileset. Time-related fields are not updated for read-only OSS filesets.

The fields in the **stat64** structure have the following meanings and content:

**st_dev** OSS device identifier for a fileset.

Values for local OSS objects are listed in the following table. Values for local Guardian objects are described in **Use on Guardian Objects**, and values for remote Guardian or OSS objects are described in **Use on Remote Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | ID of device containing directory entry |
| Directory | ID of device containing directory |
| FIFO | ID of special fileset for pipes |
| **AF_UNIX** socket | ID of device containing the fileset in which the socket file was created |
| **/dev/null** | ID of device containing directory entry |
| **/dev/tty** | ID of device containing directory entry |

**st_ino** File serial number (inode number). The file serial number and OSS device identifier uniquely identify a regular OSS file within an OSS fileset.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | File serial number (unique) |
| Directory | File serial number (unique) |
| FIFO | File serial number (unique) |
| **AF_UNIX** socket | File serial number of the socket file (unique) |
| **/dev/null** | File serial number (unique) |
| **/dev/tty** | File serial number (unique) |

The **st_ino** value for all node entries in **/E** (including the entry for the logical link from the local node name to the root fileset on the local node) is the value for the root fileset on the corresponding node. If normal conventions are followed, this value is always 0 (zero), so entries in **/E** appear to be nonunique. Values for objects on remote nodes are unique only among the values for objects within the same fileset on that node.

**st_mode** File mode. The following bits are ORed into the **st_mode** field:

**S_IFMT** File type. This field can contain one of the following values:

**S_IFCHR** Character special file.

**S_IFDIR** Directory.

**S_IFIFO** FIFO.

**S_IFREG** Regular file.

**S_IFSOCK** Socket.

For an **AF_UNIX** socket, the user permissions from the inode for the socket are returned for the permission bits. The access flags are also returned from the inode.

**S_IRWXG** Permissions for the owning group, or if the **st_acl** flag is set, permissions for the the **class** ACL entry.

**S_IRWXO** Other class

**S_IRWXU** Owner class

**S_ISGID** Set group ID on execution

**S_ISUID** Set user ID on execution

**S_ISVTX** Sticky bit; used only for directories (not ORed for files in **/G**, the Guardian file system)

Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

**st_nlink** Number of links.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Number of links to the file |
| Directory | Number of links to the directory |
| FIFO | Number of links to the file |
| **AF_UNIX** socket | Number of links to the socket file |
| **/dev/null** | Number of links to the file |
| **/dev/tty** | Number of links to the file |

**st_acl** If set to 1, indicates that the file has optional access control list (ACL) entries. For compatibility with HP-UX, the member name **st_aclv** is provided as alias for **st_acl**. For more information about access control lists, see the **acl(5)** reference page.

**st_fileprivs** File privileges. For information about file privileges see the **setfilepriv(2)** reference page.

**st_uid** User ID.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | User ID of the file owner |
| Directory | User ID of the file owner |
| FIFO | User ID of the file owner |
| **AF_UNIX** socket | User ID of the creator of the socket file |
| **/dev/null** | User ID of the super ID |
| **/dev/tty** | User ID of the super ID |

**st_gid** Group ID.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Group ID of the file group |
| Directory | Group ID of the file group |
| FIFO | Group ID of the file group |
| **AF_UNIX** socket | Group ID of the creater of the socket file |
| **/dev/null** | Group ID of the super ID |
| **/dev/tty** | Group ID of the super ID |

**st_basemode** If the **st_acl** flag is set, contains the permissions for the file owner, owning group, and others. If the **st_acl** flag is not set, **st_basemode** is 0 (zero).

**st_rdev** Remote device ID.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Undefined |
| Directory | Undefined |
| FIFO | Undefined |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | Undefined |
| **/dev/tty** | ID of the device |

**st_size** File size.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Size of the file in bytes |
| Directory | 4096 |
| FIFO | 0 (zero) |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | 0 (zero) |
| **/dev/tty** | 0 (zero) |

**st_atime** Access time.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last access |
| Directory | Time of the last access |
| FIFO | Time of the last access |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_mtime** Modification time.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last data modification |
| Directory | Time of the last modification |
| FIFO | Time of the last data modification |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_ctime** Status change time.

Values for OSS objects are listed in the following table. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Time of the last file status change |
| Directory | Time of the last file status change |
| FIFO | Time of the last file status change |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**Use on Guardian Objects**

You can use the **lstat64( )** function like the **stat64( )** or **fstat64( )** function on files in **/G**, but you cannot create symbolic links in **/G**.

The **st_dev** and **st_ino** fields of the **stat64** structure do not uniquely identify Guardian files (files in **/G**).

The **st_dev** field is unique for **/G**, for each disk volume, and for each Telserv process (or other process of subdevice type 30), because each of these is a separate fileset.

The **S_ISGUARDIANOBJECT** macro can indicate whether an object is a Guardian object when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is a Guardian object and **FALSE** otherwise.

The **st_ino** field is a nonunique encoding of the Guardian filename.

The **st_rdev** field contains a minor device number for each **pty***n* entry in **/G/ztnt/**, representing each Telserv process subdevice.

The **st_size** field of an EDIT file (file code 101) is the actual (physical) end of file, not the number of bytes in the file. For directories, **st_size** is set to 4096.

When an OSS function is called for a Guardian EDIT file, the **st_mtime** field is set to the last modification time. The **st_atime** field indicates the last time the file was opened, and the **st_ctime** field is set equal to **st_mtime**. No other time-related fields are updated by OSS functions.

The **st_ctime** and **st_atime** fields for Guardian regular disk files (except for EDIT files) are updated by OSS function calls, not by Guardian procedure calls.

The time fields for **/G**, **/G/***vol*, and **/G/***vol*/*subvol* always contain the current time.

When the *path* parameter points to the name of a Guardian process that is not a process of subtype 30, the **lstat64( )** function call fails. The value -1 is returned and **errno** is set to [ENOENT].

The **lstat64( )** function always returns access modes of "d---------" when the *path* parameter points to a Guardian subvolume that has a reserved name beginning with ZYQ. The other access modes reported for files in **/G** vary according to the file type.

The following table shows the mapping between Guardian files and their corresponding file types described in the **st_mode** field.

| Example in /G | Guardian File Type | st_mode File Type | Permissions |
|---|---|---|---|
| **/G** | N/A | Directory | r-xr-xr-x |
| *vol* | Disk volume | Directory | rwxrwxrwx |
| *vol*/*subvol* | Subvolume | Directory | rwxrwxrwx |
| *vol*/*subvol*/*fileid* | Disk file | Regular file | See following text |
| *vol*/**#123** | Temporary disk file | Regular file | See following text |
| **ztnt** | Subtype 30 process | Directory | --x--x--x |
| **ztnt/#pty0001** | Subtype 30 process with qualifier | Character special | rw-rw-rw- |
| **vol1/zyq00001** | Subvolume | Directory | --------- |

A Guardian file classified as a directory is always owned by the super ID.

Guardian permissions are mapped as follows:

- Guardian network or any user permission is mapped to OSS other permission.
- Guardian community or group user permission is mapped to OSS group permission.
- Guardian user or owner permission is mapped to OSS owner permission.
- Guardian super ID permission is mapped to OSS super ID permission.
- Guardian read permission is mapped to OSS read permission.
- Guardian write permission is mapped to OSS write permission.
- Guardian execute permission is mapped to OSS execute permission.
- Guardian purge permission is ignored.

Users are not allowed read access to Guardian processes.

OSS file permissions are divided into three groups (owner, group, and other) of three permission bits each (read, write, and execute). The OSS permission bits do not distinguish between remote and local users as Guardian security does; local and remote users are treated alike.

**Use on Remote Objects**

The content of the **st_dev** field of the **stat64** structure is unique for each node in **/E** because each node is a separate fileset. Values for directories within **/E** are the same as described for objects on the local HP NonStop node.

The **S_ISEXPANDOBJECT** macro can indicate whether an object in the **/E** directory is on a remote HP NonStop node when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is on a remote HP NonStop node and **FALSE** otherwise.

**Use From the Guardian Environment**

A Guardian process can use thee **lstat( )** function when you use the **#define _XOPEN_SOURCE_EXTENDED 1** feature test macro or an equivalent compiler command option to compile the process.

The **lstat64( )** function belongs to a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. You cannot close these file numbers by calling the Guardian FILE_CLOSE_ procedure.

- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

For J06.08 and earlier J-series RVUs, H06.19 and earlier H-series RVUs, or G-series RVUs, the OSS Network File System (NFS) cannot access OSS objects that have OSS ACLs that contain optional ACL entries.

For J06.09 and later J-series RVUs and H06.20 and later H-series RVUs, access by the OSS Network File System (NFS) to OSS objects that have OSS ACLs that contain optional ACL entries can be allowed, depending upon the NFSPERMMAP attribute value for the fileset that contains the object. For more information about NFS and ACLs, see the **acl(5)** reference page.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **lstat64( )** function sets **errno** to the corresponding value:

[EACCES] Search permission is denied for a component of the pathname pointed to by the *path* parameter.

[EFAULT] Either the *buffer* parameter or the *path* parameter points to a location outside of the allocated address space of the process.

[EFSBAD] The program attempted an operation involving a fileset with a corrupted fileset catalog.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[ENAMETOOLONG]
One of these names is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

You can call the **pathconf( )** function to obtain the applicable limits.

[ENOENT] One of the following conditions exists:

- The file specified by the *path* parameter does not exist.

- *path* points to an empty string.
- The specified pathname cannot be mapped to a valid Guardian filename.
- The specified pathname points to the name of a Guardian process that is not of subtype 30.
- The *path* parameter specifies a file on a remote HP NonStop node but communication with the remote node has been lost.

[ENOROOT] The program attempted an operation while the root fileset was unavailable.

[ENOTDIR] A component of the pathname specified by the *path* parameter is not a directory.

[ENOTSUP] The **path** parameter refers to a file on a logical disk volume administered through the Storage Management Foundation (SMF).

[ENXIO] An invalid device or address was specified during an input or output operation on a special file. One of the following events occurred:

- A device was specified that does not exist, or a request was made beyond the limits of the device.
- The fileset containing the requestor's current working directory or root directory is not mounted. This error can occur after failure and restart of an OSS name server process until the fileset has been repaired and remounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

## RELATED INFORMATION

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **chown(2)**, **fstat(2)**, **fstat64(2)**, **open(2)**, **open64(2)**, **pipe(2)**, **readlink(2)**, **stat(2)**, **stat64(2)**, **symlink(2)**, **utime(2)**.

## STANDARDS CONFORMANCE

This function is an HP extension to the XPG4 Version 2 specification.

## NAME

**mkdir -** Creates a directory

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/types.h>** /* optional except for POSIX.1 */
**#include <sys/stat.h>**

**int mkdir(**
**const char ****path***,**
**mode_t** *mode***);**

## PARAMETERS

*path* Points to the pathname for the new directory.

If any component of the *path* parameter refers to a symbolic link, the link is traversed and pathname resolution continues.

If the final component of the *path* parameter refers to an existing entity, the call fails and **errno** is set to [EEXIST].

*mode* Specifies the mask for the read, write, and search/execute (**RWX**) flags for owner, group, and others. Also specifies the file type flags for the directory.

The value of this parameter is constructed by logically ORing flags that are defined in the **sys/stat.h** header file. The permission bits are affected by the value of this parameter but depend on both the support for OSS ACLs on the system on which this process is running and on the fileset that contains the new directory. See "ACL Inheritance" in the **acl(5)** reference page.

The file type flags are described in **DESCRIPTION**.

## DESCRIPTION

The **mkdir( )** function creates a new directory with the following attributes:

- The owner ID is set to the effective user ID of the calling process. Directories within **/G** (the Guardian file system) are the exception; for them, the owner ID is set to 65535 (the super ID).

- The group ID is set to the group ID of the parent directory if the **S_ISGID** flag is set in the parent directory; otherwise, the group ID is set to the effective group ID of the calling process. Directories within **/G** (the Guardian file system) are the exception; for them, the group ID is set to 255.

- The value of the *mode* parameter is constructed by logically ORing flags that are defined in the **sys/stat.h** header file.

  If the parent directory of the created file does not have default OSS access control list (ACL) entries, the permissions for the new file are the bit-wise AND of this *mode* parameter with the complement of the process umask (see the **umask(2)** reference page). If the parent directory of the created file has default ACL entries, the permissions for the new file are affected by the value of this parameter but depend on both the support for OSS ACLs on the system on which this process is running and on the fileset that contains the new directory. See "ACL Inheritance" in the **acl(5)** reference page.

  Directories within **/G** (the Guardian file system) are the exception; for them, all the permission bits ("rwxrwxrwx") are automatically set.

- The new directory is empty except for **.** (dot) and **..** (dot-dot).

To execute the **mkdir( )** function, a process must have search permission for the parent directory of the directory pointed to by the *path* parameter and write permission in the parent directory of the *path* directory.

The **mkdir( )** function cannot create a directory named **/dev**, **/dev/tty**, or **/dev/null** in the root directory of the OSS file system. The **mkdir( )** function cannot create a directory named **lost+found** in the root directory of an OSS fileset. If these directories are missing from the system, such a function call fails and sets **errno** to the value [EPERM]. When these directories already exist (the normal case), **errno** is set to [EEXIST].

If bits in the *mode* parameter other than the file permission bits, **S_ISVTX**, or **S_IFDIR** are set, **mkdir( )** fails and sets **errno** to [EINVAL].

Upon successful completion, the **mkdir( )** function marks the **st_atime**, **st_ctime**, and **st_mtime** fields of the directory for update and marks the **st_ctime** and **st_mtime** fields of the new directory's parent directory for update.

### Use on Guardian Objects

The **mkdir( )** function succeeds within **/G** (the Guardian file system) only when creating a Guardian subvolume that is exactly three directories under the root (for example, **/G/vol/subvol**). This Guardian subvolume must be empty. If the subvolume is not empty, **errno** is set to [EEXIST]. When the call succeeds, the resulting directory (subvolume) is owned by the super ID.

### File Type Flags

The file type flags that can be logically ORed into the value specified in the *mode* parameter are as follows:

**S_IFDIR** Directory in the OSS file system or empty subvolume in **/G**, the Guardian file system.

**S_ISVTX** Sticky bit; used only for directories (cannot be used for files in **/G**, the Guardian file system).

When set, a user can remove files from the directory only if the user either:

- Has write permission for the directory and is the owner of either the directory or the file being removed
- Has appropriate privileges

### Use From the Guardian Environment

The **mkdir( )** function is one of a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **mkdir( )** function returns the value 0 (zero). If the function call fails, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the directory is not created and the **mkdir( )** function sets **errno** to the corresponding value:

[EACCES] Creating the requested directory requires writing in a directory with a mode that denies write permission, or search permission is denied on the parent directory of the directory to be created.

[EEXIST] The named file already exists.

[EFAULT] The *path* parameter is an invalid address.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EINVAL] One of the following conditions exists:

- The program supplied an invalid value for the *mode* parameter.
- The pathname supplied in the call attempts to create a directory in the Guardian file system, but the pathname cannot be mapped to a valid Guardian subvolume name.

[EIO] A physical input or output error has occurred.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of the following is true:

- A component of the prefix of the pathname pointed to by the *path* parameter does not exist.
- The *path* parameter points to an empty string.
- The *path* parameter names a symbolic link, but the file to which it refers does not exist.
- The *path* parameter specifies a file on a remote HP NonStop node but communication with the remote node has been lost.

[ENOROOT] One of the following conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node and communication with the remote name server has been lost.

[ENOSPC] The fileset does not contain enough space to hold the contents of the new directory or to extend the parent directory of the new directory.

[ENOTDIR] A component of the pathname prefix is not a directory.

[ENXIO] The fileset containing the client's current working directory or root directory is not mounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] One of the following conditions is true:

- The call attempted to create a directory named **lost+found** in the root directory of an OSS fileset, or attempted to create a directory named **/dev**, **/dev/tty**, or **/dev/null** in the the root directory of the OSS file system.
- The call attempted to create a subvolume directory in **/G** (the Guardian file system) that has a name beginning with ZYQ.
- The call attempted to create a file in the **/E** directory.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The named file resides on a read-only fileset.

**RELATED INFORMATION**

Commands: **chmod(1)**, **getacl(1)**, **mkdir(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **mknod(2)**, **rmdir(2)**, **umask(2)**.

Miscellaneous topics: **acl(5)**.

**STANDARDS CONFORMANCE**

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- If bits in the *mode* parameter other than the file permission bits, **S_ISVTX**, or **S_IFDIR** are set, **mkdir( )** fails and sets **errno** to [EINVAL].
- The group ID is set to the group ID of the parent directory if the **S_ISGID** flag is set in the parent directory; otherwise, the group ID is set to the effective group ID of the calling process. Directories within **/G** (the Guardian file system) are the exception; for them, the group ID is set to 255.

The following are extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EFSBAD], [EINVAL], [EIO], [ENOROOT], [ENXIO], [EOSSNOTRUNNING], and [EPERM] can be returned.

## NAME

**mknod** - Creates a file or assigns a pathname to a character special file

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/stat.h>**

**int mknod(**
**const char ***_path_**,**
**mode_t** _mode_**,**
**dev_t** _device_**);**

## PARAMETERS

_path_
Specifies the pathname of the new file. If the final component of the _path_ parameter names a symbolic link, the link is traversed and pathname resolution continues.

_mode_
Specifies the file type, attributes, and access permissions. This parameter is constructed by logically ORing one file type value with any valid value for an attribute for a file of that type, and with any valid access permissions.

The following file type values are supported:

**S_IFCHR** The file is a character special file.

**S_IFDIR** The file is a directory special file.

**S_IFIFO** The file is a FIFO special file.

**S_IFREG** The file is a regular file.

Values other than **S_IFIFO** can be used only if the process has appropriate privileges.

The file type value **S_IFBLK** is not supported in the OSS file system. If **S_IFBLK** is specified, the function call fails and **errno** is set to the value of [EINVAL].

The following access permissions are supported:

**S_IRGRP** Read access by members of the group list

**S_IROTH** Read access by others

**S_IRUSR** Read access by the owner of the file

**S_IRWXG** Read, write, or execute (search) access by members of the group list

**S_IRWXO** Read, write, or execute (search) access by others

**S_IRWXU** Read, write, or execute (search) access by the owner of the file

**S_ISGID** Set the group ID of the file upon execution of the file

**S_ISUID** Set the user ID of the file upon execution of the file

**S_ISVTX** Restrict the deletion of files in a directory (ignored for other file types)

**S_IWGRP** Write access by members of the group list

**S_IWOTH** Write access by others

**S_IWUSR** Write access by the owner of the file

**S_IXGRP** Execute (search) access by members of the group list

**S_IXOTH** Execute (search) access by others

**S_IXUSR** Execute (search) access by the owner of the file

*device* Specifies the type of device on which the file is created. This hexadecimal value must be 0 (zero) unless the file type **S_IFCHR** is specified for the *mode* parameter. When **S_IFCHR** is specified, the following values are valid:

**0x0000000300000000**
The device is an infinite data source or data sink, such as **/dev/null**.

**0x0000000200000000**
The device is a controlling terminal, such as **/dev/tty**.

Specifying any other value for the *device* parameter when **S_IFCHR** is specified for the *mode* parameter causes the function call to fail and **errno** to be set to [EINVAL].

## DESCRIPTION

The **mknod( )** function creates a new special or regular file. Using the **mknod( )** function to create file types other than FIFOs requires appropriate privileges.

When the **mknod( )** function creates a regular file in a fileset that supports access control lists (ACLs), and the parent directory for that file has an ACL that contains default ACL entries, the ACL for the file inherits the default ACL entries of the parent directory as actual (nondefault) ACL entries for the new file. When the **mknod( )** function creates a directory in a fileset that supports ACLs, the ACL for the new directory inherits the default ACL entries of the parent directory both as default and as actual (nondefault) ACL entries. For detailed information about ACL inheritance, see the **acl(5)** reference page.

For the **mknod( )** function to finish successfully, a process must have search permission and write permission in the parent directory of the *path* parameter.

The new file has the following characteristics:

- A file type as specified by the *mode* parameter.
- An owner ID set to the effective user ID of the process.
- A group ID set to the effective group ID of the process or to the group ID of the parent directory of the file.

- Access permission and attribute bits set according to the value of the *mode* parameter, modified as described in "ACL Inheritance" in the **acl(5)** reference page.

Upon successful completion of the function call, the **st_atime**, **st_ctime**, and **st_mtime** fields of the file are marked for update. The **st_ctime** and **st_mtime** fields of the directory that contains the new entry are also marked for update.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted by the same file permissions and owner privileges as any other user ID: It has no special privileges unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

Executable files that have the PRIVSOARFOPEN privilege and that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use From the Guardian Environment

The **mknod( )** function can be used by a Guardian process when the process has been compiled using the **#define _XOPEN_SOURCE_EXTENDED 1** feature-test macro or an equivalent compiler command option.

The **mknod( )** function is one of a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file-system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

### Use on Guardian Objects

When **S_IFREG** is specified for the *mode* parameter, the *path* parameter can be any valid version of the following:

**/G/***vol***/***subvol*
: Where *vol* already exists. If *vol* does not exist, the function call fails and **errno** is set to the value of [EINVAL].

**/G/***vol***/***subvol***/***fileid*
: Where *vol* already exists and *fileid* specifies a regular disk file (an odd unstructured Enscribe file). If *vol* does not exist, the function call fails and **errno** is set to the value of [EINVAL].

If only **/G/***vol* is specified, the function call fails and **errno** is set to the value of [EPERM].

When **S_IFCHR** is specified for the *mode* parameter, any specification for the *path* parameter that uses **/G** causes the function call to fail and **errno** to be set to [EPERM].

When **S_IFDIR** is specified for the *mode* parameter, a specification of **/G/***vol* for the *path* parameter causes the function call to fail and **errno** to be set to [EINVAL].

If any other file type value is used for the *mode* parameter of a file in **/G**, the function call fails and **errno** is set to the value of [EINVAL].

The file access permissions **S_ISUID**, **S_ISGID**, and **S_ISVTX** are ignored when you are creating files in the Guardian file system.

**NOTES**

Use the **mkfifo( )** function instead of the **mknod( )** function to create a FIFO when you need to port an application to a UNIX system that does not support XPG4 Version 2.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

**RETURN VALUES**

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the new file is not created and the **mknod( )** function sets **errno** to the corresponding value:

[EACCES] A component of the pathname prefix denies search permission, or write permission is denied on the parent directory of the file to be created.

[EEXIST] The named file exists.

[EFAULT] The *path* parameter points outside the process's allocated address space.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EINVAL] One of the following conditions exists:

- The value **S_IFBLK** was specified for the *mode* parameter.
- A value other than 0 (zero) was specified for the *device* parameter when a value other than **S_IFCHR** was specified for the *mode* parameter.
- An invalid value was specified for the *device* parameter when the value **S_IFCHR** was specified for the *mode* parameter.
- The *mode* parameter specifies a file type of **S_IFDIR** but the *path* parameter specifies a pathname of the form **/G/***vol*.
- The *mode* parameter specifies a file type of **S_IFIFO** but the *path* parameter specifies a pathname in **/G** (the Guardian file system).

[EIO] During an access of the file system, an I/O error occurred.

[ELOOP] Too many symbolic links were encountered in resolving the value of the *path* parameter.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path* parameter

- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the pathname pointed to by the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of the following conditions exists:

- The named directory does not exist.
- The specified pathname is an empty string.
- The specified pathname cannot be mapped to a valid Guardian filename.
- The *path* parameter includes a symbolic link, but the file to which it refers does not exist.
- The *path* parameter specifies a file on a remote HP NonStop node but communication with the remote node has been lost.

[ENOROOT] The root fileset (fileset 0) is not in the STARTED state.

[ENOSPC] The directory that would contain the new file cannot be extended, or the fileset is out of resources.

[ENOTDIR] A component of the pathname prefix is not a directory.

[ENXIO] The fileset containing the client's working directory or effective root directory is not mounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] One of the following conditions exists:

- The *mode* parameter specifies a file type other than **S_IFIFO** and the calling process does not have appropriate privileges.
- The *mode* parameter specifies a file type of **S_IFREG** but the *path* parameter specifies a pathname of the form **/G/***vol*.
- The *mode* parameter specifies a file type of **S_IFCHR** but the *path* parameter specifies a pathname in **/G** (the Guardian file system).
- The call attempted to create a file in the **/E** directory.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The directory in which the file is to be created is located on a read-only fileset.

## RELATED INFORMATION

Commands: **chmod(1)**, **getacl(1)**, **mkdir(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **mkdir(2)**, **mkfifo(3)**, **open(2)**, **open64(2)**, **stat(2)**, **stat64(2)**, **umask(2)**.

Miscellaneous topics: **acl(5)**.

## STANDARDS CONFORMANCE

The OSS file system does not support block special files. The file type **S_IFBLK** is therefore not valid.

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EFSBAD], [ENOROOT], [ENXIO], and [EOSSNOTRUNNING] can be returned.
- Behavior is defined when values for the *mode* parameter other than **S_IFIFO** are specified.
- Behavior is defined when values for the *device* parameter other than 0 (zero) are specified.

## NAME

**msgctl -** Performs message control operations

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

## SYNOPSIS

**#include <sys/msg.h>**

**int msgctl(**
  **int** *msqid***,**
  **int** *cmd***,**
  **struct msqid_ds** **buf***);**

## PARAMETERS

*msqid* Specifies the message queue identifier.

*cmd* Specifies the type of operation. The possible values for *cmd* and the operations they perform are as follows:

**IPC_RMID** Removes the message queue identifier and deallocates its associated **msqid_ds** structure.

This is a restricted operation. The effective user ID of the calling process either must have appropriate privileges or must be equal to the value of the owner's user ID (**msg_perm.uid** field) or the creator's user ID (**msg_perm.cuid** field) in the associated **msqid_ds** structure.

**IPC_SET** Sets the message queue identifier by copying selected values in the structure specified by the *buf* parameter into corresponding fields in the **msqid_ds** structure associated with the message queue identifier.

This is a restricted operation. The effective user ID of the calling process must have appropriate privileges or must be equal to the value of the owner's user ID (**msg_perm.uid** field) or the creator's user ID (**msg_perm.cuid** field) in the associated **msqid_ds** structure.

Only a process with appropriate privileges can increase the value of **msg_qbytes**.

**IPC_SETNONFT**
Disables fault tolerance for the message queue specified by the *msqid* parameter. The default operation of message queues makes them fault tolerant so that interprocess communication does not lose data.

This is a restricted operation. The effective user ID of the calling process must have appropriate privileges or must be equal to the value of the owner's user ID (**msg_perm.uid** field) or the creator's user ID (**msg_perm.cuid** field) in the associated **msqid_ds** structure.

**IPC_STAT** Queries the message queue identifier by copying the contents of its associated **msqid_ds** data structure into the structure specified by the *buf* parameter.

*buf* Specifies the address of a **msqid_ds** structure. This structure is used only with the **IPC_STAT** and **IPC_SET** values of the *cmd* parameter. With **IPC_STAT**, the results of the query are copied to this structure. With **IPC_SET**, the values in this structure are used to set certain fields in the **msqid_ds** structure associated with the message queue identifier. In either case, the calling process must have allocated the structure before making the call.

## DESCRIPTION

The **msgctl( )** function allows a process to query or set the contents of the **msqid_ds** structure associated with the specified message queue identifier. It also allows a process to remove the message queue identifier and its associated **msqid_ds** structure. The value of the *cmd* parameter determines which operation is performed.

The **IPC_SET** value of the *cmd* parameter uses the user-supplied contents of the *buf* parameter to set the corresponding fields of the **msqid_ds** structure associated with the message queue identifier:

- The owner's user ID field (**msg_perm.uid**) is set as specified in the input.
- The owner's group ID field (**msg_perm.gid**) is set as specified in the input.
- The access modes field (**msg_perm.mode**) is set as specified in the low-order nine bits of the corresponding field in the input.
- The maximum number of bytes field (**msg_qbytes**) for the queue is set as specified in the input.
- The field for the time of the last **msgctl( )** operation that changed the structure (**msg_ctime**) is set as specified in the input.

### Message Queue Use Between Environments

Guardian processes cannot use OSS functions for access to OSS message queues. If called from a Guardian process, this function fails and **errno** is set to [ENOTOSS].

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **msgctl( )** function sets **errno** to the value that corresponds to the condition.

[EACCES] The *cmd* parameter is **IPC_STAT**, but the calling process does not have read permission.

[EFAULT] The **msqid_ds** structure associated with the message queue identifier cannot be found.

[EINVAL] One of the following conditions exists:

- The *msqid* parameter does not specify a valid message queue identifier.
- The *cmd* parameter is not a valid command.

- All processes for the relevant message server have failed.
- The message queue corresponding to the value specified as the *msqid* parameter has been removed from the system.

[EMSGQNOTRUNNING]
The message queue server associated with the message queue identifier is not running.

[ENOMEM] Memory allocation failed and one possibility is that the amount of memory consumed by the message queues exceeds 16GB.

[ENOTOSS] The calling process is not an OSS process. The requested operation is not supported from the Guardian environment.

[EPERM] One of the following conditions exists:

- The *cmd* parameter is equal to either **IPC_RMID** or **IPC_SET**, and the calling process does not have appropriate privileges.
- The *cmd* parameter is equal to **IPC_SET**, and an attempt is being made to increase the value of the **msg_qbytes** field when the effective user ID of the calling process does not have appropriate privileges.

## RELATED INFORMATION

Functions: **msgget(2)**, **msgrcv(2)**, **msgsnd(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The **IPC_SETNONFT** value for the *cmd* parameter is supported.
- The **errno** values [EFAULT], [EMSGQNOTRUNNING], [ENOMEM], and [ENOTOSS] can be returned.

## NAME

**msgget -** Creates or returns the identifier for a message queue

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/yosskdll**

## SYNOPSIS

**#include <sys/msg.h>**

**int msgget(**
    **key_t** *key***,**
    **int** *msgflg***);**

## PARAMETERS

*key*
: Specifies the key that identifies the message queue. The **IPC_PRIVATE** key can be used to ensure the return of a new (unused) message queue identifier.

*msgflg*
: Specifies the following creation flag values:

**IPC_CREAT** If the key does not exist, the **msgget( )** function creates a message queue identifier using the given key.

**IPC_CREAT | IPC_EXCL**
If the key already exists, the **msgget( )** function fails and returns an error notification.

## DESCRIPTION

The **msgget( )** function returns the message queue identifier for the message queue identified by the *key* parameter. If the *key* parameter already has a message queue identifier associated with it and (*msgflg* **& IPC_CREAT**) is 0 (zero), that identifier is returned.

A new message queue identifier and its associated data structure are created when either of the following is true:

- The value of **IPC_PRIVATE** is used for the *key* parameter.
- The *key* parameter does not already have a message queue identifier associated with it, and (*msgflg* **& IPC_CREAT**) is not 0 (zero).

After creating a new message queue identifier, the **msgget( )** function initializes the **msqid_ds** structure associated with the identifier as follows:

- The **msg_perm.cuid** and **msg_perm.uid** fields are set to the effective user ID of the calling process.
- The **msg_perm.cgid** and **msg_perm.gid** fields are set to the effective group ID of the calling process.
- The low-order nine bits of the **msg_perm.mode** field are set to the low-order nine bits of *msgflg*.
- The **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** fields are all set to 0 (zero).

- The **msg_ctime** field is set to the current time. This field is updated when any of the following events occur:
  - — The message queue identifier is created.
  - — The message queue identifier is removed.
- The **msg_qbytes** field is set to the system limit.

The message queue identifier is used for the following purposes:

- It identifies a specific message server.
- It allows detection of references to a previously removed message queue.
- It allows detection of attempts to reference message queues in other processors.

### Key Creation

The key represents a user-designated name for a given message queue. Keys are usually selected by calling the **ftok( )** function before calling the **msgget( )** function. The **ftok( )** function returns a key based on a path and an interprocess communications identifier. This key is passed to the **msgget( )** function, which returns a message queue identifier.

The message queue identifier is then used in calls to the **msgctl( )**, **msgrcv( )**, and **msgsnd( )** functions.

### Uniqueness of Identifiers

The system recycles no-longer-used message queue identifiers after a long time elapses.

### Processor or Disk Process Failures

If a processor fails and if its OSS message-queue server was running as a process pair, no queued messages are lost. The backup server process takes over, and there are no effects on the successful completion of this function.

If the OSS message-queue server was not running as a process pair when its processor failed, queued messages are lost and the function call fails with **errno** set to [EINVAL]. Thereafter, a process cannot successfully call any function using the associated message queue identifier.

### Cleaning Up Message Queue Identifiers

A message queue identifier remains allocated until it is removed. An allocated message queue identifier is not removed when the last process using it terminates. The user must remove allocated message queue identifiers that are not attached to processes to avoid wasting system resources.

The status of message queue identifiers can be checked with the **ipcs** command. Message queue identifiers can be removed using the **ipcrm** command. The associated data structure is removed only after the final detach operation.

### Message Queue Use Between Environments

Guardian processes cannot use OSS functions for access to OSS message queues. Such a call fails, and **errno** is set to [ENOTOSS].

## RETURN VALUES

Upon successful completion, a message queue identifier is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **msgget( )** function sets **errno** to the value that corresponds to the condition.

[EACCES] A message queue identifier exists for the *key* parameter but operation permission, which is specified by the low-order nine bits of the *msgflg* parameter, is not granted.

[EEXIST] A message queue identifier exists for the *key* parameter, and both **IPC_CREAT** and **IPC_EXCL** are set.

[EFAULT] The **msqid_ds** structure associated with the message queue identifier cannot be found.

[EINVAL] One of the following conditions exists:

- All processes for the relevant message server have failed.
- The message queue corresponding to the message queue identifier associated with the *key* parameter has been removed from the system.

[EMSGQNOTRUNNING]
The message queue server associated with the message queue identifier is not running.

[ENOENT] A message queue identifier does not exist for the *key* parameter and the **IPC_CREAT** value is not set.

[ENOMEM] Memory allocation failed and one possibility is that the amount of memory consumed by the message gueues exceeds 16GB.

[ENOSPC] A message queue identifier cannot be created because the system-imposed limit on the maximum number of allowed message queue identifiers would be exceeded.

[ENOTOSS] The calling process is not an OSS process. The requested operation cannot be performed from the Guardian environment.

## RELATED INFORMATION

Commands: **ipcrm(1)**, **ipcs(1)**.

Functions: **ftok(3)**, **msgctl(2)**, **msgrcv(2)**, **msgsnd(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EINVAL], [EMSGQNOTRUNNING], [ENOMEM], and [ENOTOSS] can be returned.

## NAME

**msgrcv** - Receives a message from a message queue

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/yosskdll**

## SYNOPSIS

```
#include <sys/msg.h>

int msgrcv(
        int msqid,
        void *msgp,
        size_t msgsz,
        long int msgtyp,
        int msgflg);
```

## PARAMETERS

*msqid* Specifies the identifier of the message queue from which to read a message.

*msgp* Specifies a pointer to the **msgbuf** structure that is to receive the message. (See the **NOTES** section.)

*msgsz* Specifies the maximum number of bytes allowed for the received message.

*msgtyp* Specifies the message type to read from the queue.

*msgflg* Specifies the following actions:

- The action to be taken by the system if there are no *msgtyp* messages in the queue.
- Whether to truncate the message if its length exceeds the value specified by *msgsz*.
- Whether the *msgp* parameter is formatted for 32-bit and 64-bit interoperability. (See the **NOTES** section for detailed information on how to use the **MSG_32BIT_MTYPE** flag and **msgbuf** structure to allow 32-bit and 64-bit processes to communicate using one **msgbuf** structure.)

## DESCRIPTION

The **msgrcv( )** function reads a message from the queue associated with the *msqid* parameter. It returns the number of bytes in the received message.

The *msgp* parameter points to a user-defined **msgbuf** structure. The structure receives the message read from the queue.

The *msgsz* parameter specifies the maximum size allowed for the received message. If the message is longer than the value specified by the *msgsz* parameter, the system takes action based on the use of the **MSG_NOERROR** flag in the *msgflg* parameter.

The *msgtyp* parameter specifies the message type that the process wants to receive. Possible values and their results are as follows:

0 (zero) The process receives the message at the head of the queue.

> 0 (positive) The process receives the first message of the requested message type.

< 0 (negative) The process receives the first message of the lowest type on the queue. To qualify as the lowest type, a message's type must be less than or equal to the absolute value of the *msgtyp* parameter.

The *msgflg* parameter specifies actions that the system should take:

- If the **IPC_NOWAIT** flag is used and the queue does not contain a message of the requested type, the function call returns immediately with the value -1 and **errno** is set to [ENOMSG].
- If the **IPC_NOWAIT** flag is not used and the queue does not contain a message of the requested type, the system suspends the calling process. The process remains suspended until one of the following occurs:
  - — A message of the requested type appears in the queue. In this case, the system wakes the process to receive the message.
  - — The specified message queue identifier is removed from the system. In this case, the system sets **errno** to [EIDRM] and returns the value -1 to the calling process.
  - — The process catches a signal. In this case, the process does not receive the message; instead, it resumes execution as directed by a **sigaction( )** function call.
- If the **MSG_NOERROR** flag is used and the message is longer than the value specified by the *msgsz* parameter, the system truncates the message to *msgsz* bytes and discards the truncated portion without notifying the calling process.
- If the **MSG_NOERROR** flag is not used and the message is longer than the value specified by the *msgsz* parameter, the system returns an **errno** value of [E2BIG] to the calling process and leaves the message in the queue.
- If the **MSG_32BIT_MTYPE** flag is logically ORed with *msgflg* and the caller is a 64-bit process, the function assumes the **mtype** field in the **struct msgbuf** pointed to by *msgp* is of type **int** instead of type **long**. (See the **NOTES** section for detailed information on how to use the **MSG_32BIT_MTYPE** flag and **msgbuf** structure to allow 32-bit and 64-bit processes to communicate using one **msgbuf** structure.)
- If the **MSG_32BIT_MTYPE** flag is not present, the function assumes the **mtype** field in the **struct msgbuf** pointed to by *msgp* is of type **long int**. (The type **long int** is a different size for 32-bit processes and 64-bit processes.)

### Message Queue Use Between Environments

Guardian processes cannot use OSS functions to access OSS message queues. If called from a Guardian process, the function call fails and **errno** is set to [ENOTOSS].

## NOTES

The **IPC_NOWAIT** flag is defined in the **sys/ipc.h** header file.

The user-supplied **msgbuf** structure, used to store received messages, can be defined as follows:

```
struct msgbuf {
        long int mtype;
        char mtext[ ];
};
```

The **mtype** field is set to the message type assigned by the sender.

The **mtext** field is set to the message text. The message size is less than or equal to the value of the *msgsz* parameter specified in the last successful call to **msgrcv( )**.

However, the data type for the **mtype** field (**long int**) is a problem when 64-bit processes send and receive messages with 32-bit processes because the length of the field varies depending on whether the caller is a 32-bit or 64-bit process. This field is 32 bits for 32-bit processes and 64 bits for 64-bit processes.

Because the **msgbuf** structure is user supplied, it is the application's responsibility to handle the differences in data types between 32-bit and 64-bit senders and receivers. For example, suppose a 64-bit process sends a **msgbuf** structure that contains a 64-bit **mtype** field to a 32-bit process. The 32-bit process that receives the structure does not understand how to process the message because it expects this field to be only 32 bits long. Additionally, although the **mtext** field starts at the 65th bit of the message, the 32-bit process expects the **mtype** field to start at the 33rd bit of the message.

To allow interoperability between 64-bit and 32-bit processes, it is recommended that 64-bit applications define their **msgbuf** structure as follows:

```
struct msgbuf {
        int mtype;
        char mtext[ ];
};
```

and that all 64-bit callers use the **MSG_32BIT_MTYPE** flag in the *msgflg* parameter for all calls to **msgrcv( )** and **msgsnd( )**.

## RETURN VALUES

Upon successful completion, the **msgrcv( )** function returns the number of bytes actually stored in the **mtext** field. Also, the system updates the **msqid_ds** structure associated with the message queue identifier as follows:

- Decrements the value in the **msg_qnum** field by 1.
- Decrements the value in the **msg_cbytes** field by the message text size.
- Sets the **msg_lrpid** field to the OSS process ID of the calling process.
- Sets the **msg_rtime** field to the current time.

When the **msgrcv( )** function fails, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **msgrcv( )** function sets **errno** to the value that corresponds to the condition.

[E2BIG] The number of bytes to be received in the **mtext** field is greater than the value of the *msgsz* parameter and the **MSG_NOERROR** flag is used in the *msgflg* parameter.

[EACCES] The calling process does not have read permission for the specified message queue.

[EFAULT] The **msqid_ds** structure associated with the message queue identifier cannot be found.

[EIDRM] The message queue identified by the *msqid* parameter has been removed from the system.

[EINTR] The operation was interrupted by a signal.

[EINVAL] One of the following conditions exists:

- The *msqid* parameter does not specify a valid message queue identifier.
- The value of the *msgsz* parameter is less than 0 (zero) or greater than the system-defined limit.
- All processes for the relevant message server have failed.
- Both **MSG_32BIT_MTYPE** and **MSG_64BIT_MTYPE** are specified in the *msgflag* parameter.

[EMSGQNOTRUNNING]
The message queue server associated with the message queue identifier is not running.

[ENOMEM] Memory allocation failed and one possibility is that the amount of memory consumed by the message queues exceeds 16GB.

[ENOMSG] The queue does not contain a message of the requested type and the **IPC_NOWAIT** flag is used in the *msgflg* parameter.

[ENOTOSS] The calling process is not an OSS process. The requested operation cannot be performed from the Guardian environment.

## RELATED INFORMATION

Functions: **msgctl(2)**, **msgget(2)**, **msgsnd(2)**, **sigaction(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EMSGQNOTRUNNING], [ENOMEM], and [ENOTOSS] can be returned.

## NAME

**msgsnd -** Sends a message to a message queue

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/yosskdll**

## SYNOPSIS

**#include <sys/msg.h>**

**int msgsnd(**
**int** *msqid***,**
**const void *** *msgp***,**
**size_t** *msgsz***,**
**int** *msgflg***);**

## PARAMETERS

*msqid* Specifies the identifier of the message queue in which to place the message. The identifier is typically returned by a previous call to the **msgget( )** function.

*msgp* Specifies a pointer to the **msgbuf** structure that contains the message. (See the **NOTES** section.)

*msgsz* Specifies the size of the data array in the **msgbuf** structure.

*msgflg* Specifies the following actions:

- The action that the system should take if either or both of the following are true (the system runs out of internal buffer space):
  - — The current number of bytes in the message queue is equal to **msg_qbytes** (in the **msqid_ds** structure).
  - — The total number of messages in all message queues is equal to the system-defined limit.
- Whether the *msgp* parameter is formatted for 32-bit and 64-bit interoperability. (See the **NOTES** section for detailed information on how to use the **MSG_32BIT_MTYPE** flag and **msgbuf** structure to allow 32-bit and 64-bit processes to communicate using one **msgbuf** structure.)

## DESCRIPTION

The **msgsnd( )** function sends a message to the queue associated with the *msqid* parameter.

The *msgp* parameter points to a user-defined **msgbuf** structure. The structure identifies the message type and contains a data array with the message text.

The size of the data array is specified by the *msgsz* parameter. The *msgsz* value can be from 0 (zero) through a system-defined maximum.

The *msgflg* parameter specifies actions that the system should take:

- If the **IPC_NOWAIT** flag is used in the *msgflg* parameter and the system runs out of internal buffer space, the system does not send the message and returns to the calling process immediately.

- If the **IPC_NOWAIT** flag is not used in the *msgflg* parameter the system runs out of internal buffer space, the system suspends the calling process. The process remains suspended until one of the following occurs:
  - — The blocking condition is removed. In this case, the system sends the message.
  - — The specified message queue identifier is removed from the system. In this case, the system sets **errno** to [EIDRM] and returns the value -1 to the calling process.
  - — The process catches a signal. In this case, the message is not sent and the process resumes execution as directed by a **sigaction( )** function call.
- If the **MSG_32BIT_MTYPE** flag is logically ORed with *msgflg* and the caller is a 64-bit process, the function assumes the **mtype** field in the **struct msgbuf** pointed to by *msgp* is of type **int** instead of type **long**. (See the **NOTES** section for detailed information on how to use the **MSG_32BIT_MTYPE** flag and **msgbuf** structure to allow 32-bit and 64-bit processes to communicate using one **msgbuf** structure.)
- If the **MSG_32BIT_MTYPE** flag is not present, the function assumes the **mtype** field in the **struct msgbuf** pointed to by *msgp* is of type **long int**. (The type **long int** is a different size for 32-bit processes and 64-bit processes.)

If the **msgsnd( )** function finishes successfully, the system updates the **msqid_ds** structure associated with the *msqid* parameter. Specifically, it does the following:

- Increments the value in the **msg_qnum** field by 1.
- Increments the value in the **msg_cbytes** field by the message text size.
- Sets the **msg_lspid** field to the OSS process ID of the calling process.
- Sets the **msg_stime** field to the current time.

### Message Queue Use Between Environments

Guardian processes cannot use OSS functions to access OSS message queues. If called from a Guardian process, the function call fails and **errno** is set to [ENOTOSS].

## NOTES

The **IPC_NOWAIT** flag is defined in the **sys/ipc.h** header file.

The user-supplied **msgbuf** structure can be defined as follows:

```
struct msgbuf {
        long int mtype;
        char mtext[ ];
};
```

The **mtype** field is a user-chosen positive integer that represents the message type. A receiving process can use the message type to select only those messages it wants to receive from the queue. (See the **msgrcv(2)** reference page.)

The **mtext** field contains any text of the length specified by the *msgsz* parameter.

However, the data type for the **mtype** field (**long int**) is a problem when 64-bit processes send and receive messages with 32-bit processes because the length of the field varies depending on whether the caller is a 32-bit or 64-bit process. This field is 32 bits for 32-bit processes and 64 bits for 64-bit processes.

Because the **msgbuf** structure is user supplied, it is the application's responsibility to handle the differences in data types between 32-bit and 64-bit senders and receivers. For example, suppose a

64-bit process sends a **msgbuf** structure that contain a 64-bit **mtype** field to a 32-bit process. The 32-bit process that receives the structure does not understand how to process the message because it expects this field to be only 32 bits long. Additionally, although the **mtext** field starts at the 65th bit of the message, the 32-bit process expects the **mtype** field to start at the 33rd bit of the message.

To allow interoperability between 64-bit and 32-bit processes, it is recommended that 64-bit applications define their **msgbuf** structure as follows:

```
struct msgbuf {
        int mtype;
        char mtext[ ];
};
```

and that all 64-bit callers use the **MSG_32BIT_MTYPE** flag in the *msgflg* parameter for all calls to **msgrcv( )** and **msgsnd( )**.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **msgsnd( )** function sets **errno** to the value that corresponds to the condition.

[EACCES] The calling process does not have the correct access permission for the operation.

[EAGAIN] The **IPC_NOWAIT** flag is used in the *msgflg* parameter, and either the maximum number of message headers has been allocated or the size of the message exceeds the amount of space currently available on the target queue.

[EFAULT] The **msqid_ds** structure associated with the message queue identifier cannot be found.

[EIDRM] The message queue identified by the *msqid* parameter has been removed from the system.

[EINTR] The operation was interrupted by a signal.

[EINVAL] One of the following conditions is true:

- The *msqid* parameter does not specify a valid message queue identifier.
- The value of the **mtype** field is less than 1.
- The value of the *msgsz* parameter is less than 0 (zero) or greater than the value defined by the **MAXMSG** value in SCF.
- All processes for the relevant message server have failed.
- Both **MSG_32BIT_MTYPE** and **MSG_64BIT_MTYPE** are specified in the *msgflag* parameter.

[EMSGQNOTRUNNING]
The message queue server associated with the message queue identifier is not running.

[ENOMEM] Memory allocation failed and one possibility is that the amount of memory consumed by the message queues exceeds 16GB.

[ENOTOSS] The calling process is not an OSS process. The requested operation cannot be performed from the Guardian environment.

**RELATED INFORMATION**

Functions: **msgctl(2)**, **msgget(2)**, **msgrcv(2)**, **sigaction(2)**.

**STANDARDS CONFORMANCE**

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EMSGQNOTRUNNING], [ENOMEM], and [ENOTOSS] can be returned.

# Section 5. System Functions (n - p)

This section contains reference pages for Open System Services (OSS) system function calls with names that begin with **n** through **p**. These reference pages reside in the **cat2** directory and are sorted alphabetically by U.S. English conventions in this section.

## NAME

**nice -** Changes the scheduling priority of the calling process

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series native Guardian processes: implicit libraries
H-series OSS processes: implicit libraries

## SYNOPSIS

**#include <unistd.h>**

**int nice(**
    **int** *increment***);**

## PARAMETERS

*increment* Specifies a value that is added to the current **nice** value of the calling process.

The **nice** value of the calling process is maintained by the system and affects the scheduling priority of the process. Increasing the **nice** value lowers the scheduling priority of the process. Decreasing the **nice** value increases the scheduling priority of the process.

If the value specified for *increment* increases the **nice** value of the calling process such that it exceeds the maximum value possible for **nice**, **nice** is set to its maximum value.

A negative value can be specified for *increment* if the process has appropriate privileges.

If the value specified for *increment* decreases the **nice** value of the calling process such that it becomes less than the minimum value possible for **nice**, **nice** is set to its minimum value.

## DESCRIPTION

The **nice( )** function increases or decreases the **nice** value of the calling process.

The **nice** value is a nonnegative number in the range 0 through (2***NZERO -**1). **NZERO** is defined in the **limits.h** header file.

The **nice** value is a relative value for scheduling priority among executing processes.

The **nice** value is an attribute of a process in both the Guardian and OSS environments. The default value of **nice** for a newly created process is the value defined for **NZERO** in the **limits.h** header file. The **nice** value affects scheduling priority but does not determine scheduling priority.

### Use on Guardian Objects

The **nice( )** function can only be used by a process on itself.

The Guardian priority of a process after a call to the **nice( )** function is calculated as follows:

```
New Guardian priority = old Guardian priority
        - (new nice value - old nice value)
```

which is the same as:

```
New Guardian priority = old Guardian priority
        - ((old nice value + increment) - old nice value)
```

If the sum of the old **nice** value and the *increment* is

- less than 0 (zero), then the new **nice** value is 0 (zero).
- greater than 39, then the new **nice** value is 39 because the current value of 2***NZERO** -1 is 39.

Refer to **NOTES** for a description of the relative priorities of Guardian and OSS processes.

### Use From the Guardian Environment

The **nice( )** function can be used from the Guardian environment.

## NOTES

Changing the Guardian priority of a process does not affect the **nice** value of the process.

The **nice** value of a process can also be changed by a call to the Guardian procedure PROCESS_SETINFO_. The **nice** value can be determined by a call to the Guardian procedure PROCESS_GETINFOLIST_. Refer to the *Guardian Procedure Calls Reference Manual* for additional information.

The **nice** value is not the value used by the operating system to compare scheduling priorities among processes in all environments.

The scheduling priority for processes running in the Guardian environment is defined as increasing as the priority number increases. This convention is the opposite of the convention used on UNIX systems, where a lower priority number means a higher scheduling priority.

Processes running in the OSS environment have their scheduling priorities determined using UNIX conventions. The OSS priority of a process after a call to the **nice( )** function is calculated as follows:

```
New OSS priority = 199 - new Guardian priority
        - new nice value
```

which is the same as:

```
New OSS priority = 199 - new Guardian priority
        - (old nice value + increment)
```

## RETURN VALUES

Upon successful completion, the **nice( )** function returns the new **nice** value minus the value of **NZERO**. If the function call fails, the value -1 is returned, the **nice** value for the process is not changed, and **errno** is set to indicate the error.

Because a value of -1 also can be returned by a successful completion of the function call, an application program that needs to check for failure of the function call should set **errno** to 0 (zero) before calling the **nice( )** function.

## ERRORS

If the following condition occurs, **nice( )** sets **errno** to the corresponding value:

[EPERM] The calling process specified a negative value for the *increment* parameter but does not have appropriate privileges.

## RELATED INFORMATION

Functions: **execl(2)**, **execle(2)**, **execlp(2)**, **execv(2)**, **execve(2)**, **execvp(2)**, **fork(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**.

## NAME

**open** - Opens a file for reading or writing; creates a regular file in the OSS environment

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <sys/stat.h>      /* optional except for POSIX.1 */
#include <fcntl.h>

int open(
        const char *path,
        int oflag
        [ , mode_t mode ]);
```

## PARAMETERS

*path* Points to the pathname of the file to be opened or created.

You cannot specify the files **/lost+found**, **/dev**, **/dev/tty**, and **/dev/null** for this parameter when the **O_CREAT** flag is set for the *oflag* parameter. Attempts to create these files cause the function call to fail and **errno** to be set to [EINVAL].

If the *path* parameter refers to a symbolic link, the **open( )** function opens the file pointed to by the symbolic link.

If the *path* parameter refers to a file in the Guardian file system (**/G**), additional restrictions apply. See the subsection **Opening Guardian Files** in the **DESCRIPTION** section of this reference page for more information.

*oflag* Specifies the type of access, special open processing, the type of update, and the initial state of the open file. The parameter value is constructed by logically ORing special open processing flags. These flags are defined in the **fcntl.h** header file and are described in **DESCRIPTION**.

*mode* Specifies the read, write, and execute permissions of the file and the file type flags for the file.

This parameter is required if the file does not exist and the **O_CREAT** flag is set in the *oflag* parameter. If the file already exists and **O_CREAT** is set, this parameter is required and must have a valid value, but this parameter has no effect on the file (you cannot use this parameter to change the permissions of the file).

If this parameter is specified when values other than **O_CREAT** are used in the *oflag* parameter, the values specified for *mode* have no effect on whether the file is opened for reading or writing.

The value of this parameter is constructed by logically ORing flags that are defined in the **sys/stat.h** header file.

If the parent directory of the created file does not have default OSS access control list (ACL) entries, the permissions for the new file are the bit-wise AND of this *mode* parameter with the complement of the process umask (see the **umask(2)** reference page). If the parent directory of the created file has default ACL entries, the permissions for the new file are affected by the value of this parameter but depend on both the support for OSS ACLs on the system on which this process is running and on the fileset that contains the new directory. See "ACL Inheritance" in the **acl(5)** reference page.

If a file opened for writing has file privileges such as PRIVSOARFOPEN or PRIVSETID, these file privileges are removed. Only Members of Safeguard SECURITY-PRV-ADMINISTRATOR (SEC-PRIV-ADMIN or SPA) group are permitted to explicitly set file privileges. File privileges can be set using the **setfilepriv( )** function or the **setfilepriv** command only. See also "Considerations for Restricted-Access Filesets."

The file type flags are described in **DESCRIPTION**.

## DESCRIPTION

This function can open:

- OSS files up to a size limit of approximately 2 gigabytes
- Guardian Format 1 files up to a size limit of approximately 2 gigabytes
- Guardian Format 2 files up to a size limit of approximately 2 gigabytes

For information about opening larger files, see the **open64(2)** reference page.

The **open( )** function establishes a connection between the file indicated by the *path* parameter and the returned file descriptor. Subsequent I/O function calls, such as **read( )** and **write( )**, use the opened file descriptor to access that file.

The returned file descriptor is the lowest-numbered file descriptor not currently open for that process. A corresponding Guardian environment file number is also assigned.

The file offset, marking the current position within the file, is set to the beginning of the file. The new file descriptor is set to remain open across the processing of any of the **exec** or **tdm_exec** set of functions. (See the **fcntl(2)** reference page.)

The file status flags and file access flags are designated by the *oflag* parameter. The *oflag* parameter is constructed by a bitwise-inclusive-OR of exactly one of the file access flags (**O_RDONLY**, **O_WRONLY**, or **O_RDWR**) with one or more of the file status flags.

You cannot use the **open( )** function to create a First-in, First-out (FIFO) special file. Use the **mkfifo( )** function instead.

### File Access Flags

The file access flags are:

**O_RDONLY** The file is open only for reading.

**O_WRONLY** The file is open only for writing.

**O_RDWR** The file is open for reading and writing.

You must specify exactly one of the file access flags.

**File Status Flags**

The file status flags that specify special open processing are:

**O_CREAT** Create and open the file. If the file exists, this flag has no effect except as noted under the **O_EXCL** flag. If the file does not exist, a regular file is created with these characteristics:

- If access control lists (ACLs) are supported, ACL entries are added to the file ACL as described in "ACL Inheritance" in the **acl(5)** reference page.
- The owner ID of the file is set to the effective user ID of the process.
- The group ID of the file is determined by the value of the **S_ISGID** flag in the parent directory. If **S_ISGID** is set, the group ID of the file is set to the group ID of the parent directory; otherwise, the group ID of the file is set to the effective group ID of the calling process. If the file is a Guardian file (that is, within **/G**), the group ID is set to that of the primary group of the effective user ID.
- The file permission and attribute bits are set to the value of the *mode* parameter, modified as listed:
  - The file permission bits are set as described in "ACL Inheritance" in the **acl(5)** reference page.
  - The set user ID attribute (**S_ISUID** bit) is cleared.
  - The set group ID attribute (**S_ISGID** bit) is cleared.

  If bits other than the file permission and appropriate file-type bits are set in the *mode* parameter, **errno** is set to [EINVAL].

**O_EXCL** Open the file in exclusive access mode.

If the file exists and the **O_EXCL** and **O_CREAT** flags are set, the open fails. If the file exists and the **O_EXCL** flag is set and the **O_CREAT** flag is not set, the open succeeds.

**O_NOCTTY** Open the file but not as a controlling terminal. If the *path* parameter identifies a terminal device, this flag ensures that the terminal device does not become the controlling terminal for the process.

When opening a file that is not a terminal device, the **O_NOCTTY** flag is ignored.

**O_TRUNC** Open the file and empty it. If the file does not exist or if the file is not a regular file, this flag has no effect. If the file exists and is a regular file, and if the file is successfully opened with either read/write access or write-only access:

- The length of the file is truncated to 0 (zero).
- The owner and group of the file are unchanged.
- The set user ID attribute of the file mode is cleared.

The open fails if any of these conditions is true:

- The file supports enforced record locks, and another process has locked a portion of the file.
- The file does not allow write access.
- The *oflag* parameter also specifies the **O_RDONLY** flag.

If the *oflag* parameter also specifies the **O_SYNC** flag, the truncation is a synchronous update.

A program can request some control over when updates should be made permanent for a regular file opened for write access.

The file status flags that define the initial state of the open file are:

**O_APPEND** Open the file only for append access. If set, the file pointer is set to the end of the file before each write.

This flag is ignored for Telserv terminal devices.

**O_NONBLOCK**
Open the file for nonblocked access. If set, the call to **open( )** does not block, and subsequent **read( )** or **write( )** operations on the file are nonblocking.

When opening a regular disk file or an OSS directory, the **O_NONBLOCK** flag is ignored.

Calling the **open( )** function with the **O_NONBLOCK** flag for FIFO files and for character special devices that support nonblocking opens is supported.

Calling the **open( )** function with the **O_NONBLOCK** flag is supported for Telserv terminal devices (**tty**) as listed:

- For a static window, the open operation is always allowed; it finishes when the connection is established.
- For a dynamic window, the open operation is allowed only if a connection is already established.

Calling the **open( )** function with the **O_NONBLOCK** flag is supported for OSSTTY terminal devices (**ztty**). OSSTTY devices support only three static windows, one each for **#stdin**, **#stdout**, and **#stderr**.

**O_SYNC** The **O_SYNC** flag provides a high level of data integrity for writes to regular files. For HP NonStop systems, you can use the OSS Monitor to select one of multiple levels of fault tolerance. For more information, see the discussion of the FTIOMODE attribute in the *Open System Services Management and Operations Guide.*

### General Notes on *oflag* Parameter Flag Values

The effect of setting the **O_CREAT** flag is immediate.

When opening a file with the **O_CREAT** flag set:

- If the named file does not already exist, a regular disk file is created.

- If the named file is not a regular file, the **O_CREAT** flag is ignored.

When opening a FIFO file with the **O_RDONLY** flag set:

- If the **O_NONBLOCK** flag is not set, the **open( )** function blocks until another process opens the file for writing. If the file is already open for writing (even by the calling process), the function returns without delay.
- If the **O_NONBLOCK** flag is set, the **open( )** function returns immediately.

When opening a FIFO file with the **O_WRONLY** flag set:

- If the **O_NONBLOCK** flag is not set, the **open( )** function blocks until another process opens the file for reading. If the file is already open for reading (even by the calling process), the function returns without delay.
- If the **O_NONBLOCK** flag is set, the **open( )** function returns an error if no process currently has the file open for reading.

The **O_RDWR** file access flag is supported when opening a FIFO file; the call to the **open( )** function finishes immediately, even if the **O_NONBLOCK** flag is not set.

When opening a character special file that supports nonblocking opens, such as a terminal device:

- If the **O_NONBLOCK** flag is not set, the **open( )** function blocks until the device is ready or available.
- If the **O_NONBLOCK** flag is set, the **open( )** function returns without waiting for the device to be ready or available. Subsequent behavior of the device is device-specific.

When opening a directory, the open fails, and **errno** is set to [EISDIR], if either of these conditions is true:

- The directory is **/E** or **/G** (the Guardian file system) or a directory within **/G**.
- The directory is not **/E** or **/G** and is not within **/E** or **/G**, and the file access flag is either **O_WRONLY** or **O_RDWR**.

### File Type Flags

The file type flags that can be logically ORed into the value specified in the *mode* parameter are:

**S_IFREG** Regular file in the OSS file system or in **/G**, the Guardian file system.

**S_ISVTX** Sticky bit; used only for directories (cannot be used for files in **/G**, the Guardian file system).

**S_NONSTOP** **S_NONSTOP** is an alias for **O_SYNC**.

### Opening Guardian Files

If the file is a Guardian file (that is, if it is in the **/G** file system):

- The file can be opened only if it is:
  - — A Format 1 file or a Format 2 file that is smaller than 2 gigabytes, on a physical disk volume, and either:
    - — An odd, unstructured Enscribe file. In this case, it is opened as a regular file with a primary and secondary extent size that is a multiple of 2. If the extent size is odd, the open fails.

      If the unstructured buffer size was not 4096, a successful open makes the buffer size 4096 (as if the Guardian procedure SETMODE was called for mode 93 with a parameter value of 4096).
    - — An EDIT file (file code 101). In this case, it is opened as a regular file for read-only access.
  - — A Telserv or OSSTTY terminal process.

  You cannot use the **open( )** function on any other type of Guardian object. An attempt to open:

  - — A Format 2 file that is larger than approximately 2 gigabytes fails with **errno** set to [EOVERFLOW].
  - — A structured file fails with **errno** set, usually to [EINVAL].
  - — A file administered through the Storage Management Foundation (SMF) fails with **errno** set to [ENOTSUP].
  - — Any file or device of any other type not described here fails with **errno** set, usually to [EINVAL].

  An attempt to open a volume, a subvolume, or a process other than a TTY simulation process (**/G/***vol*, **/G/***vol*/*subvol*, or **/G/***process*, respectively) fails with **errno** set to [EISDIR].

- An attempt to open a subvolume with a reserved name beginning with ZYQ (for example, **/G/vol2/zyq00004**) fails with **errno** set to [EACCES].
- An attempt to open a file within a subvolume with a reserved name beginning with ZYQ (for example, **/G/vol2/zyq00004/z000002x**) fails with **errno** set to [EACCES].
- If the file is not an EDIT file (that is, the file code is not 101), it is opened in shared exclusion mode.
- If the file is an EDIT file and read-only access is specified, the file is opened in protected exclusion mode in the Guardian environment.
- If the file is an EDIT file and write access is specified, the call fails with **errno** set to [EINVAL].
- The **sysconf( )** function reports the maximum number of opens as the upper limit of opens per process. The actual limit depends on other factors, such as the size of the process file segment (PFS) and the number of existing opens on directories or on files in the Guardian environment.

- If the open requires file creation, the Guardian file created will be Format 1, odd, unstructured, and file code 180.
- If the open requires file creation, the file is given access permissions compatible with the standard security permissions for the Guardian creator access ID (CAID) of the calling process.

During **open( )** processing, all access permissions are checked. This includes Guardian environment checks by Guardian standard security mechanisms (and by the Safeguard product) for Guardian disk file and process access.

### Considerations for Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted by the same file permissions and owner privileges as any other user ID.

Executable files that have the PRIVSETID file privilege and that are started by super ID can perform privileged switch ID operations (such as by using the **setuid( )** function) to switch to another ID and then access files in restricted-access filesets as that ID. Executable files without the PRIVSETID file privilege that perform privileged switch ID operations are unconditionally denied access to restricted-access filesets.

Executable files that have the PRIVSOARFOPEN privilege and that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

If a file opened for writing has file privileges such as PRIVSOARFOPEN or PRIVSETID, these file privileges are removed. Only Members of Safeguard SECURITY-PRV-ADMINISTRATOR (SEC-PRIV-ADMIN or SPA) group are permitted to explicitly set file privileges. File privileges can be set using the **setfilepriv( )** function or the **setfilepriv** command only.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use From the Guardian Environment

A call to the **open( )** function in the Guardian environment requires an OSS pathname and returns an OSS file-system file descriptor, regardless of the file system containing the file.

The **open( )** function belongs to a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file-system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. You cannot close these file numbers by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the function returns the file descriptor, a nonnegative integer. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the function sets **errno** to the corresponding value:

[EACCES] One of these conditions exists:

- Search permission is denied on a component of the pathname prefix.
- The type of access specified by the *oflag* parameter is denied for the named file.
- The file does not exist, and write permission is denied for the parent directory.
- The **O_TRUNC** flag is specified, and write permission is denied.
- The process attempted to open a Guardian subvolume with a reserved name beginning with ZYQ or a file within such a subvolume.
- The process attempted to open a static Telserv window that is not yet connected.

[EEXIST] The **O_CREAT** and **O_EXCL** flags are set, and the named file exists.

[EFAULT] The *path* parameter is an invalid address.

[EFILEBAD] One of these conditions exists:

- The function call attempted to open a Guardian EDIT file, but the structure of the file is bad.
- The function call attempted to open a Guardian EDIT file, but the corrupted flag is set in the file label.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EGUARDIANOPEN]
The function call attempted to open a Guardian EDIT file for write access or for Guardian shared or exclusive exclusion access, but the file has already been opened with a Guardian procedure call.

[EINTR] A signal was caught during the open operation. This value is returned only for character special files (terminal devices) and for FIFO special files.

[EINVAL] One of these conditions exists:

- The call attempted to create a directory named **lost+found** in the root directory of an OSS fileset, or it attempted to create a directory named **/dev**, **/dev/tty**, or **/dev/null** in the root directory of the OSS file system.

- The function call specified the **O_CREAT** flag but did not specify the *mode* parameter.
- The **O_CREAT** flag is set and bits other than the file permission and appropriate file type flags are set in the *mode* parameter.
- Both the **O_TRUNC** flag and **O_RDONLY** flag are set.
- None of the access flags **O_RDONLY**, **O_WRONLY**, or **O_RDWR** are set.
- The function call attempted to create a Guardian file (that is, a file in the **/G** file system), but the pathname cannot be mapped to a valid Guardian filename.
- The function call attempted to open a Guardian file of a type other than those permitted.
- The function call attempted to create a Guardian temporary file.

[EIO] A physical input or output error occurred. The device where the file is stored might be in the down state, or both processors that provide access to the device might have failed.

Data might have been lost during transfer.

[EISDIR] One of these conditions exists:

- The named file is an OSS directory, and write access is requested.
- The named file is a Guardian directory (**/G** or a directory in the **/G** file system).

[ELOOP] Too many symbolic links were encountered in translating the *path* parameter.

[EMFILE] The system limit for open file descriptors per process has reached the maximum permitted.

[ENAMETOOLONG]
One of these names is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

You can call the **pathconf( )** function to obtain the applicable limits.

[ENETDOWN]
The call was blocked during access to a FIFO, and communication has been lost with the remote node containing the other end of the FIFO.

[ENFILE] One of these conditions exists:

- The maximum number of file descriptors of this file type (socket, pipe, etc.) for this processor are already open.

- The limit for open file descriptors of this file type has not been exceeded, but the maximum number of all file descriptors for this processor are already open.

[ENOENT] One of these conditions exists:

- The **O_CREAT** flag is not set, and the named file does not exist.
- **O_CREAT** is set, and the pathname prefix does not exist.
- The *path* parameter points to an empty string.
- The function call attempted to open a file in the Guardian file system, but the specified pathname cannot be mapped to a valid Guardian filename.
- The *path* parameter points to a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOROOT] One of these conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node, and communication with the remote name server has been lost.

[ENOSPC] The directory that would contain the new file cannot be extended, the file does not exist, and the **O_CREAT** flag is set.

[ENOTDIR] A component of the pathname prefix is not a directory.

[ENOTSUP] The *path* parameter specifies a Guardian file on an SMF logical volume and one of the following conditions exists:

- The local system is running an RVU prior to J06.15 or H06.26.
- The *path* parameter specifies a file in /E and the remote system is running an RVU prior to J06.15 or H06.26.

[ENXIO] One of these conditions exists:

- The named file is a character special file, and the device associated with this special file does not exist.
- The **O_NONBLOCK** flag is set, the named file is a FIFO file, the **O_WRONLY** flag is set, and no process has the file open for reading.
- The fileset containing the client's current working directory or root directory is not mounted.

[EOPNOTSUPP]
The named file is a socket bound to the file system (not an **AF_INET** or **AF_INET6** socket) and cannot be opened.

[EOSSNOTRUNNING]
A required system process is not running.

[EOVERFLOW]
The file size is larger than approximately 2 gigabytes.

[EPERM] One of these conditions exists:

- The function call attempted to create a file named **lost+found** in the root directory of an OSS fileset.
- The call attempted to create a file in **/E**.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The named file resides on a read-only fileset, and write access is required.

[ETXTBSY] The file is being executed, and the *oflag* value is **O_WRONLY** or **O_RDWR**.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **close(2)**, **creat(2)**, **creat64(2)**, **fcntl(2)**, **lseek(2)**, **lseek64(2)**, **mknod(2)**, **open64(2)**, **read(2)**, **stat(2)**, **stat64(2)**, **umask(2)**, **write(2)**.

Miscellaneous topics: **acl(5)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- The **O_RDWR** flag is supported for FIFO files.
- The group ID of the new file is determined by the value of the **O_ISGID** flag in the parent directory.
- The **O_NONBLOCK** flag is ignored for regular disk files and directory files.
- The **O_NOCTTY** flag is ignored for regular disk files and directory files.
- The **O_CREAT** flag is ignored for FIFOs and tty files.
- If the **O_CREAT** flag is specified and bits other than the file permission and appropriate file type flags are set in the *mode* parameter, **errno** is set to [EINVAL].
- If the **O_TRUNC** flag is specified and the **O_RDONLY** access flag is specified, the open fails.
- The **O_TRUNC** flag is ignored for files other than regular files.
- Attempting to open an OSS directory with an access flag of **O_WRONLY** or **O_RDWR** fails.

- Specifying the **O_NONBLOCK** flag when opening character special devices that support nonblocking opens is supported.

HP extensions to the XPG4 Version 2 specification are:

- Opening Guardian files (that is, files in the **/G** file system) is supported, as described under **Opening Guardian Files** in **DESCRIPTION**.
- Access control lists (ACLs) for OSS files are supported.
- The **errno** values [EFAULT], [EFILEBAD], [EFSBAD], [EGUARDIANOPEN], [EIO], [ELOOP], [ENETDOWN], [ENOTSUP], [EOSSNOTRUNNING], and [EPERM] can be returned.

## NAME

**open64 -** Opens a file for reading or writing; creates a regular file in the OSS environment

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/types.h>** /* optional except for POSIX.1 */
**#include <sys/stat.h>** /* optional except for POSIX.1 */
**#include <fcntl.h>**

**int open64(**
**const char ***path**,**
**int** *oflag*
[ **, mode_t** *mode* ]**);**

## PARAMETERS

*path* Points to the pathname of the file to be opened or created.

You cannot specify the files **/lost+found**, **/dev**, **/dev/tty**, and **/dev/null** for this parameter when the **O_CREAT** flag is set for the *oflag* parameter. Attempts to create these files cause the function call to fail and **errno** to be set to [EINVAL].

If the *path* parameter refers to a symbolic link, the **open64( )** function opens the file pointed to by the symbolic link.

If the *path* parameter refers to a file in the Guardian file system (**/G**), additional restrictions apply. See the subsection **Opening Guardian Files** in the **DESCRIPTION** section of this reference page for more information.

*oflag* Specifies the type of access, special open processing, the type of update, and the initial state of the open file. The parameter value is constructed by logically ORing special open processing flags. These flags are defined in the **fcntl.h** header file and are described in **DESCRIPTION**.

*mode* Specifies the read, write, and execute permissions of the file and the file type flags for the file.

This parameter is required if the file does not exist and the **O_CREAT** flag is set in the *oflag* parameter. If the file already exists and **O_CREAT** is set, this parameter is required and must have a valid value, but this parameter has no effect on the file (you cannot use this parameter to change the permissions of the file).

If this parameter is specified when values other than **O_CREAT** are used in the *oflag* parameter, the values specified for *mode* have no effect on whether the file is opened for reading or writing.

The value of this parameter is constructed by logically ORing flags that are defined in the **sys/stat.h** header file. If the parent directory of the created file does not have default OSS access control list (ACL) entries, the permissions for the new file are the bit-wise AND of this *mode* parameter with the complement of the process umask (see the **umask(2)** reference page). If the parent directory of the created file has default ACL entries, the permissions for the new file are affected by the value of this parameter but depend on both the support for OSS ACLs on the system on which this process is running and on the fileset that contains the new directory. See "ACL Inheritance" in the **acl(5)** reference page.

The file type flags are described in **DESCRIPTION**.

**DESCRIPTION**

The **open64( )** function is similar to the **open( )** function except that, in addition to supporting smaller files, the **open64( )** function supports:

- OSS files larger than approximately 2 gigabytes, up to a limit of approximately 1 terabyte (constrained by the space available on the disk volume)
- Both Guardian Format 1 and Guardian Format 2 files, up to the limit described in the *Open System Services Management and Operations Guide*

An application can explicitly call this function when you compile the applicaton using the **#define _LARGEFILE64_SOURCE 1** feature test macro or an equivalent compiler command option.

An application call to **creat( )** is automatically mapped to this function when you compile the application using the **#define _FILE_OFFSET_BITS 64** feature test macro or an equivalent compiler command option.

The **open64( )** function establishes a connection between the file indicated by the *path* parameter and the returned file descriptor. Subsequent I/O function calls, such as **read( )** and **write( )**, use the opened file descriptor to access that file.

The returned file descriptor is the lowest-numbered file descriptor not currently open for that process. A corresponding Guardian environment file number is also assigned.

The file offset, marking the current position within the file, is set to the beginning of the file. The new file descriptor is set to remain open across the processing of any of the **exec** or **tdm_exec** set of functions. (See the **fcntl(2)** reference page.)

The file status flags and file access flags are designated by the *oflag* parameter. The *oflag* parameter is constructed by a bitwise-inclusive-OR of exactly one of the file access flags (**O_RDONLY**, **O_WRONLY**, or **O_RDWR**) with one or more of the file status flags.

You cannot use the **open64( )** function to create a First-in, First-out (FIFO) special file. Use the **mkfifo( )** function instead.

**File Access Flags**

The file access flags are:

**O_RDONLY** The file is open only for reading.

**O_WRONLY** The file is open only for writing.

**O_RDWR** The file is open for reading and writing.

You must specify exactly one of the file access flags.

**File Status Flags**

The file status flags that specify special open processing are:

**O_CREAT** Create and open the file. If the file exists, this flag has no effect except as noted under the **O_EXCL** flag. If the file does not exist, a regular file is created with these characteristics:

- If access control lists (ACLs) are supported, ACL entries are added to the file ACL as described in "ACL Inheritance" in the **acl(5)** reference page.
- The owner ID of the file is set to the effective user ID of the process.
- The group ID of the file is determined by the value of the **S_ISGID** flag in the parent directory. If **S_ISGID** is set, the group ID of the file is set to the group ID of the parent directory; otherwise, the group ID of the file is set to the effective group ID of the calling process. If the file is a Guardian file (that is, within **/G**), the group ID is set to that of the primary group of the effective user ID.
- The file permission and attribute bits are set to the value of the *mode* parameter, modified as listed:
  - The file permission bits are set as described in "ACL Inheritance" in the **acl(5)** reference page.
  - The set user ID attribute (**S_ISUID** bit) is cleared.
  - The set group ID attribute (**S_ISGID** bit) is cleared.

  If bits other than the file permission and appropriate file-type bits are set in the *mode* parameter, **errno** is set to [EINVAL].

**O_EXCL** Open the file in exclusive access mode.

If the file exists and the **O_EXCL** and **O_CREAT** flags are set, the open fails. If the file exists and the **O_EXCL** flag is set and the **O_CREAT** flag is not set, the open succeeds.

**O_NOCTTY** Open the file but not as a controlling terminal. If the *path* parameter identifies a terminal device, this flag ensures that the terminal device does not become the controlling terminal for the process.

When opening a file that is not a terminal device, the **O_NOCTTY** flag is ignored.

**O_TRUNC** Open the file and empty it. If the file does not exist or if the file is not a regular file, this flag has no effect. If the file exists and is a regular file, and if the file is successfully opened with either read/write access or write-only access:

- The length of the file is truncated to 0 (zero).
- The owner and group of the file are unchanged.
- The set user ID attribute of the file mode is cleared.

The open fails if any of these conditions is true:

- The file supports enforced record locks, and another process has locked a portion of the file.
- The file does not allow write access.
- The *oflag* parameter also specifies the **O_RDONLY** flag.

If the *oflag* parameter also specifies the **O_SYNC** flag, the truncation is a synchronous update.

A program can request some control over when updates should be made permanent for a regular file opened for write access.

The file status flags that define the initial state of the open file are:

**O_APPEND** Open the file only for append access. If set, the file pointer is set to the end of the file before each write.

This flag is ignored for Telserv terminal devices.

**O_NONBLOCK**
Open the file for nonblocked access. If set, the call to **open64( )** does not block, and subsequent **read( )** or **write( )** operations on the file are nonblocking.

When opening a regular disk file or an OSS directory, the **O_NONBLOCK** flag is ignored.

Calling the **open64( )** function with the **O_NONBLOCK** flag for FIFO files and for character special devices that support nonblocking opens is supported.

Calling the **open64( )** function with the **O_NONBLOCK** flag is supported for Telserv terminal devices (**tty**) as listed:

- For a static window, the open operation is always allowed; it finishes when the connection is established.
- For a dynamic window, the open operation is allowed only if a connection is already established.

Calling the **open64( )** function with the **O_NONBLOCK** flag is supported for OSSTTY terminal devices (**ztty**). OSSTTY devices support only three static windows, one each for **#stdin**, **#stdout**, and **#stderr**.

**O_SYNC** The **O_SYNC** flag provides a high level of data integrity for writes to regular files. For HP NonStop systems, you can use the OSS Monitor to select one of multiple levels of fault tolerance. For more information, see the discussion of the FTIOMODE attribute in the *Open System Services Management and Operations Guide.*

### General Notes on *oflag* Parameter Flag Values

The effect of setting the **O_CREAT** flag is immediate.

When opening a file with the **O_CREAT** flag set:

- If the named file does not already exist, a regular disk file is created.
- If the named file is not a regular file, the **O_CREAT** flag is ignored.

When opening a FIFO file with the **O_RDONLY** flag set:

- If the **O_NONBLOCK** flag is not set, the **open64( )** function blocks until another process opens the file for writing. If the file is already open for writing (even by the calling process), the function returns without delay.
- If the **O_NONBLOCK** flag is set, the **open64( )** function returns immediately.

When opening a FIFO file with the **O_WRONLY** flag set:

- If the **O_NONBLOCK** flag is not set, the **open64( )** function blocks until another process opens the file for reading. If the file is already open for reading (even by the calling process), the function returns without delay.
- If the **O_NONBLOCK** flag is set, the **open64( )** function returns an error if no process currently has the file open for reading.

The **O_RDWR** file access flag is supported when opening a FIFO file; the call to the **open64( )** function finishes immediately, even if the **O_NONBLOCK** flag is not set.

When opening a character special file that supports nonblocking opens, such as a terminal device:

- If the **O_NONBLOCK** flag is not set, the **open64( )** function blocks until the device is ready or available.
- If the **O_NONBLOCK** flag is set, the **open64( )** function returns without waiting for the device to be ready or available. Subsequent behavior of the device is device-specific.

When opening a directory, the open fails, and **errno** is set to [EISDIR], if either of these conditions is true:

- The directory is **/E** or **/G** (the Guardian file system) or a directory within **/G**.
- The directory is not **/E** or **/G** and is not within **/E** or **/G**, and the file access flag is either **O_WRONLY** or **O_RDWR**.

### File Type Flags

The file type flags that can be logically ORed into the value specified in the *mode* parameter are:

**S_IFREG** Regular file in the OSS file system or in **/G**, the Guardian file system.

**S_ISVTX** Sticky bit; used only for directories (cannot be used for files in **/G**, the Guardian file system).

**S_NONSTOP** **S_NONSTOP** is an alias for **O_SYNC**.

**Opening Guardian Files**

If the file is a Guardian file (that is, if it is in the **/G** file system):

- The file can be opened only if it is:
  - — A file on a physical disk volume and either:
    - — An odd, unstructured Enscribe file. In this case, it is opened as a regular file with a primary and secondary extent size that is a multiple of 2. If the extent size is odd, the open fails.

      If the unstructured buffer size was not 4096, a successful open makes the buffer size 4096 (as if the Guardian procedure SETMODE was called for mode 93 with a parameter value of 4096).
    - — An EDIT file (file code 101). In this case, it is opened as a regular file for read-only access.
  - — A Telserv or OSSTTY terminal process.

  You cannot use the **open64( )** function on any other type of Guardian object. An attempt to open:

  - — A structured file fails with **errno** set, usually to [EINVAL].
  - — A file administered through the Storage Management Foundation (SMF) fails with **errno** set to [ENOTSUP].
  - — Any file or device of any other type not described here fails with **errno** set, usually to [EINVAL].

  An attempt to open a volume, a subvolume, or a process other than a TTY simulation process (**/G/***vol*, **/G/***vol*/*subvol*, or **/G/***process*, respectively) fails with **errno** set to [EISDIR].
- An attempt to open a subvolume with a reserved name beginning with ZYQ (for example, **/G/vol2/zyq00004**) fails with **errno** set to [EACCES].
- An attempt to open a file within a subvolume with a reserved name beginning with ZYQ (for example, **/G/vol2/zyq00004/z000002x**) fails with **errno** set to [EACCES].
- If the file is not an EDIT file (that is, the file code is not 101), it is opened in shared exclusion mode.
- If the file is an EDIT file and read-only access is specified, the file is opened in protected exclusion mode in the Guardian environment.
- If the file is an EDIT file and write access is specified, the call fails with **errno** set to [EINVAL].
- The **sysconf( )** function reports the maximum number of opens as the upper limit of opens per process. The actual limit depends on other factors, such as the size of the process file segment (PFS) and the number of existing opens on directories or on files in the Guardian environment.
- If the open requires file creation, the Guardian file created will be Format 2, odd, unstructured, and file code 180.
- If the open requires file creation, the Guardian file created is given access permissions compatible with the standard security permissions for the Guardian creator access ID

(CAID) of the calling process.

During **open64( )** processing, all access permissions are checked. This includes Guardian environment checks by Guardian standard security mechanisms (and by the Safeguard product) for Guardian disk file and process access.

### Considerations for Restricted-Access Filesets

See the **open(2)** reference page.

### Use From the Guardian Environment

A call to the **open64( )** function in the Guardian environment requires an OSS pathname and returns an OSS file-system file descriptor, regardless of the file system containing the file.

The **open64( )** function belongs to a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file-system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. You cannot close these file numbers by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the function returns the file descriptor, a nonnegative integer. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the function sets **errno** to the corresponding value:

[EACCES] One of these conditions exists:

- Search permission is denied on a component of the pathname prefix.
- The type of access specified by the *oflag* parameter is denied for the named file.
- The file does not exist, and write permission is denied for the parent directory.
- The **O_TRUNC** flag is specified, and write permission is denied.
- The process attempted to open a Guardian subvolume with a reserved name beginning with ZYQ or a file within such a subvolume.
- The process attempted to open a static Telserv window that is not yet connected.

[EEXIST] The **O_CREAT** and **O_EXCL** flags are set, and the named file exists.

[EFAULT] The *path* parameter is an invalid address.

[EFILEBAD] One of these conditions exists:

- The function call attempted to open a Guardian EDIT file, but the structure of the file is bad.
- The function call attempted to open a Guardian EDIT file, but the corrupted flag is set in the file label.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EGUARDIANOPEN]
The function call attempted to open a Guardian EDIT file for write access or for Guardian shared or exclusive exclusion access, but the file has already been opened with a Guardian procedure call.

[EINTR] A signal was caught during the open operation. This value is returned only for character special files (terminal devices) and for FIFO special files.

[EINVAL] One of these conditions exists:

- The call attempted to create a directory named **lost+found** in the root directory of an OSS fileset, or it attempted to create a directory named **/dev**, **/dev/tty**, or **/dev/null** in the root directory of the OSS file system.
- The function call specified the **O_CREAT** flag but did not specify the *mode* parameter.
- The **O_CREAT** flag is set and bits other than the file permission and appropriate file type flags are set in the *mode* parameter.
- Both the **O_TRUNC** flag and **O_RDONLY** flag are set.
- None of the access flags **O_RDONLY**, **O_WRONLY**, or **O_RDWR** are set.
- The function call attempted to create a Guardian file (that is, a file in the **/G** file system), but the pathname cannot be mapped to a valid Guardian filename.
- The function call attempted to open a Guardian file of a type other than those permitted.
- The function call attempted to create a Guardian temporary file.

[EIO] A physical input or output error occurred. The device where the file is stored might be in the down state, or both processors that provide access to the device might have failed.

Data might have been lost during transfer.

[EISDIR] One of these conditions exists:

- The named file is an OSS directory, and write access is requested.

- The named file is a Guardian directory (**/G** or a directory in the **/G** file system).

[ELOOP] Too many symbolic links were encountered in translating the *path* parameter.

[EMFILE] The system limit for open file descriptors per process has reached the maximum permitted.

[ENAMETOOLONG]
One of these names is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

You can call the **pathconf( )** function to obtain the applicable limits.

[ENETDOWN]
The call was blocked during access to a FIFO, and communication has been lost with the remote node containing the other end of the FIFO.

[ENFILE] One of these conditions exists:

- The maximum number of file descriptors of this file type (socket, pipe, etc.) for this processor are already open.
- The limit for open file descriptors of this file type has not been exceeded, but the maximum number of all file descriptors for this processor are already open.

[ENOENT] One of these conditions exists:

- The **O_CREAT** flag is not set, and the named file does not exist.
- **O_CREAT** is set, and the pathname prefix does not exist.
- The *path* parameter points to an empty string.
- The function call attempted to open a file in the Guardian file system, but the specified pathname cannot be mapped to a valid Guardian filename.
- The *path* parameter points to a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOROOT] One of these conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node, and communication with the remote name server has been lost.

[ENOSPC] The directory that would contain the new file cannot be extended, the file does not exist, and the **O_CREAT** flag is set.

[ENOTDIR] A component of the pathname prefix is not a directory.

[ENOTSUP] The *path* parameter specifies a Guardian file on an SMF logical volume and one of the following conditions exists:

- The local system is running an RVU prior to J06.15 or H06.26.
- The *path* parameter specifies a file in /E and the remote system is running an RVU prior to J06.15 or H06.26.

[ENXIO] One of these conditions exists:

- The named file is a character special file, and the device associated with this special file does not exist.
- The **O_NONBLOCK** flag is set, the named file is a FIFO file, the **O_WRONLY** flag is set, and no process has the file open for reading.
- The fileset containing the client's current working directory or root directory is not mounted.

[EOPNOTSUPP]
The named file is a socket bound to the file system (not an **AF_INET** or **AF_INET6** socket) and cannot be opened.

[EOSSNOTRUNNING]
A required system process is not running.

[EPERM] One of these conditions exists:

- The function call attempted to create a file named **lost+found** in the root directory of an OSS fileset.
- The call attempted to create a file in **/E**.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The named file resides on a read-only fileset, and write access is required.

[ETXTBSY] The file is being executed, and the *oflag* value is **O_WRONLY** or **O_RDWR**.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

**RELATED INFORMATION**

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **close(2)**, **creat(2)**, **creat64(2)**, **fcntl(2)**, **lseek(2)**, **lseek64(2)**, **mknod(2)**, **read(2)**, **stat(2)**, **stat64(2)**, **umask(2)**, **write(2)**.

Miscellaneous topics: **acl(5)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- The **O_RDWR** flag is supported for FIFO files.
- The group ID of the new file is determined by the value of the **O_ISGID** flag in the parent directory.
- The **O_NONBLOCK** flag is ignored for regular disk files and directory files.
- The **O_NOCTTY** flag is ignored for regular disk files and directory files.
- The **O_CREAT** flag is ignored for FIFOs and tty files.
- If the **O_CREAT** flag is specified and bits other than the file permission and appropriate file type flags are set in the *mode* parameter, **errno** is set to [EINVAL].
- If the **O_TRUNC** flag is specified and the **O_RDONLY** access flag is specified, the open fails.
- The **O_TRUNC** flag is ignored for files other than regular files.
- Attempting to open an OSS directory with an access flag of **O_WRONLY** or **O_RDWR** fails.
- Specifying the **O_NONBLOCK** flag when opening character special devices that support nonblocking opens is supported.

HP extensions to the XPG4 Version 2 specification are:

- Opening Guardian files (that is, files in the **/G** file system) is supported, as described under **Opening Guardian Files** in **DESCRIPTION**.
- Access control lists (ACLs) for OSS files are supported.
- The **errno** values [EFAULT], [EFILEBAD], [EFSBAD], [EGUARDIANOPEN], [EIO], [ELOOP], [ENETDOWN], [EOSSNOTRUNNING], and [EPERM] can be returned.

## NAME

**pipe -** Creates an interprocess communication channel

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <unistd.h>**

**int pipe(**
  **int** *filedes* **[2]);**

## PARAMETERS

*filedes* Specifies the address of an array of two integers into which new file descriptors are placed.

## DESCRIPTION

The **pipe**( ) function creates an interprocess channel called a pipe and returns two file descriptors in the parameters *filedes***[0]** and *filedes***[1]**. The file descriptor *filedes***[0]** is opened for reading, and the file descriptor *filedes***[1]** is opened for writing. Their integer values are the two lowest available at the time of the call to the **pipe( )** function. The **O_NONBLOCK** flag is cleared on both file descriptors. (The **fcntl( )** function can be used to set the **O_NONBLOCK** flag.)

Upon successful completion, the **pipe( )** function marks the **st_atime**, **st_ctime**, and **st_mtime** fields of the pipe for update.

The **FD_CLOEXEC** flag is cleared on both file descriptors.

### Use From the Guardian Environment

The **pipe( )** function is one of a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. If the **pipe**( ) function fails, the value -1 is returned, and **errno** is set to indicate the error.

### ERRORS

If any of these conditions occurs, the **pipe**( ) function sets **errno** to the corresponding value:

[EFAULT] The *filedes* parameter is an invalid address.

[EMFILE] No more file descriptors are available for this process.

[ENFILE] One of these conditions exists:

- The maximum number of file descriptors of this file type (socket, pipe, etc.) for this processor are already open.
- The limit for open file descriptors of this file type has not been exceeded, but the maximum number of all file descriptors for this processor are already open.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOROOT] The function was called while the root fileset (fileset 0) was not available.

[EOSSNOTRUNNING]
The function was called while a required system process was not running.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

### RELATED INFORMATION

Functions: **fcntl(2)**, **read(2)**, **select(2)**, **write(2)**.

Commands: **sh(1)**.

### STANDARDS CONFORMANCE

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EFAULT], [ENOROOT], and [EOSSNOTRUNNING] can be returned.

## NAME

**pthread_atfork -** Declares fork-handler routines to be called when the calling thread's process forks a child process

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_atfork(**
  **void (****prepare***) (void),**
  **void (****parent***) (void),**
  **void (****child***) (void));**

## PARAMETERS

*prepare* Specifies the address of a routine that performs the fork preparation handling. This routine is called in the parent process before the child process is created.

*parent* Specifies the address of a routine that performs the fork parent handling. This routine is called in the parent process after the child process is created and before the return to the caller of **fork( )**.

*child* Specifies the address of a routine that performs the fork child handling. This routine is called in the child process before the return to the caller of **fork( )**.

## DESCRIPTION

This function allows a main program or library to control resources during a **fork( )** operation by declaring fork-handler routines, as follows:

- The fork-handler routine specified by the *prepare* parameter is called before **fork( )** executes.
- The fork-handler routine specified by the *parent* parameter is called after **fork( )** executes within the parent process.
- The fork-handler routine specified by the *child* parameter is called in the new child process after **fork( )** executes.

Your program (or library) can use fork handlers to ensure that program context in the child process is consistent and meaningful. After **fork( )** executes, only the calling thread exists in the child process, and the state of all memory in the parent process is replicated in the child process, including the states of any mutexes, condition variables, and so on.

For example, in the new child process there might exist locked mutexes that are copies of mutexes that were locked in the parent process by threads that do not exist in the child process. Therefore, any associated program state might be inconsistent in the child process.

The program can avoid this problem by calling **pthread_atfork( )** to provide routines that acquire and release resources that are critical to the child process. For example, the *prepare* handler should lock all mutexes that you want to be usable in the child process. The *parent* handler just unlocks those mutexes. The *child* handler also unlocks them all — and might also create threads or reset any program state for the child process.

If no fork handling is desired, you can set any of this function's parameters to **NULL**.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## EXAMPLES

If your library uses a mutex **my_mutex**, you might provide **pthread_atfork( )** handler routines coded as follows:

```
void  my_prepare(void)
    {
    pthread_mutex_lock(&my_mutex);
    }
void  my_parent(void)
```

```
        {
            pthread_mutex_unlock(&my_mutex);
            }
        void  my_child(void)
            {
            pthread_mutex_unlock(&my_mutex);
            /* Reinitialize state that doesn't apply...like heap owned */
            /* by other threads           */
            }
          {
            .
            .
            .
           pthread_atfork(my_prepare, my_parent, my_child);
            .
            .
           fork();
          }
```

**NOTES**

Do not call **pthread_atfork( )** from within a fork-handler routine. Doing so could cause a deadlock.

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [ENOMEM] | Insufficient table space exists to record the fork-handler routines' addresses. |

**RELATED INFORMATION**

Functions: **pthread_create(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_destroy -** Destroys a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

```
int pthread_attr_destroy(
        pthread_attr_t *attr);
```

## PARAMETERS

*attr* Specifies the thread attributes object to be destroyed.

## DESCRIPTION

This function destroys a thread attributes object. Call this function when a thread attributes object will no longer be referenced.

Threads that were created using this thread attributes object are not affected by the destruction of this thread attributes object.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *attr* parameter is invalid. |

## RELATED INFORMATION

Functions: **pthread_attr_init(2)**, **pthread_create(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_getdetachstate -** Obtains the detachstate attribute of a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_attr_getdetachstate(**
    **const pthread_attr_t ****attr***,**
    **int ****detachstate***);**

## PARAMETERS

*attr* Specifies the address of the thread attributes object whose **detachstate** attribute is obtained.

*detachstate* Receives the value of the **detachstate** attribute.

## DESCRIPTION

This function obtains the value of the **detachstate** attribute of the thread attributes object specified by the *attr* parameter and returns it in the *detachstate* parameter. This attribute specifies whether threads created using the specified thread attributes object are created in a detached state.

See the **pthread_attr_setdetachstate(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the **detachstate** attribute.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

### RETURN VALUES

On successful completion, this function returns a zero and the **detachstate** attribute value is returned in *detachstate*. The attribute value **PTHREAD_CREATE_JOINABLE** indicates the thread is not detached, and the attribute value **PTHREAD_CREATE_DETACHED** indicates the thread is detached.

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The *attr* parameter does not refer to an existing thread attributes object. |

### RELATED INFORMATION

Functions: **pthread_attr_init(2)**, **pthread_attr_setdetachstate(2)**.

### STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_getguardsize -** Obtains the **guardsize** attribute of a thread attributes object

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**int pthread_attr_getguardsize(**
    **const pthread_attr_t** ***attr*,**
    **size_t** ***guardsize*);**

## PARAMETERS

*attr* Specifies the address of the thread attributes object whose **guardsize** attribute is obtained.

*guardsize* Receives the value of the **guardsize** attribute.

## DESCRIPTION

The **pthread_attr_getguardsize** function obtains the value of the **guardsize** attribute of the thread attributes object specified by the *attr* parameter and returns it in the *guardsize* parameter. The specified thread attributes object must already be initialized when this function called. The value returned for the *guardsize* parameter is either the guard size specified by the previous **pthread_attr_setguardsize** function call if there was one, or the default guard size.

When creating a thread, use a thread attributes object to specify nondefault values for thread attributes. The **guardsize** attribute of a thread attributes object specifies the minimum size (in bytes) of the guard area for the stack of a new thread.

A guard area can help a multi-threaded program detect overflow of a thread's stack. A guard area is a region of no-access memory that the system allocates at the overflow end of the thread's stack. When any thread attempts to access a memory location within this region, a memory addressing violation occurs.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## NOTES

The value of the **guardsize** attribute of a particular thread attributes object does not necessarily correspond to the actual size of the guard area of any existing thread in a multi-threaded program.

This function is not supported with the Standard POSIX Threads (SPT) library. SPT-based applications should use the **pthread_attr_getguardsize_np( )** function instead.

For detailed information about writing multi-threaded applications for the Open System Services environment using the POSIX User Thread Model library, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *attr* parameter is invalid. |

## RELATED INFORMATION

Functions: **pthread_attr_init(2)**, **pthread_attr_getguardsize_np(2)**, **pthread_attr_setguardsize(2)**.

## STANDARDS CONFORMANCE

This function conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**pthread_attr_getguardsize_np -** Obtains the guardsize attribute of a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int pthread_attr_getguardsize_np(**
    **const pthread_attr_t** ***attr*,**
    **size_t** ***guardsize* );**

## PARAMETERS

*attr* specifies the address of the thread attributes object whose **guardsize** attribute is obtained.

*guardsize* receives the value of the **guardsize** attribute.

## DESCRIPTION

This function obtains the value of the **guardsize** attribute of the thread attributes object specified by the *attr* parameter and returns it in the *guardsize* parameter. The specified thread attributes object must already be initialized when this function called.

When creating a thread, use a thread attributes object to specify nondefault values for thread attributes. The **guardsize** attribute of a thread attributes object specifies the minimum size (in bytes) of the guard area for the stack of a new thread.

A guard area can help a multithreaded program detect overflow of a thread's stack. A guard area is a region of no-access memory that the system allocates at the overflow end of the thread's stack. When any thread attempts to access a memory location within this region, a memory addressing violation occurs.

## NOTES

The value of the **guardsize** attribute of a particular thread attributes object does not necessarily correspond to the actual size of the guard area of any existing thread in a multithreaded program.

Use of this function makes your application nonportable.

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

0 Successful completion.

[EINVAL] The value specified by the *attr* parameter is invalid.

## RELATED INFORMATION

Functions: **pthread_attr_init(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification and to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

## NAME

**pthread_attr_getinheritsched -** Obtains the inherit scheduling attribute of a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

```
#include <pthread.h> | #include <spthread.h>
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

int pthread_attr_getinheritsched(
        const pthread_attr_t *attr,
        int *inheritsched);
```

## PARAMETERS

*attr* Specifies the address of the thread attributes object whose **inherit** scheduling attribute is obtained.

*inheritsched* Receives the value of the **inherit** scheduling attribute.

## DESCRIPTION

This function obtains the value of the **inherit** scheduling attribute of the thread attributes object specified by the *attr* parameter and returns it in the *inheritsched* parameter. The **inherit** scheduling attribute specifies whether threads created using the specified threads attributes object inherit the scheduling attributes of the creating thread or use the scheduling attributes stored in the threads attributes object specified by the **pthread_create( )** *attr* parameter.

See the **pthread_attr_setinheritsched(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the **inherit** scheduling attribute.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *attr* parameter is invalid. |

## RELATED INFORMATION

Functions: **pthread_attr_init(2)**, **pthread_attr_setinheritsched(2)**, **pthread_create(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_getschedparam -** Obtains the scheduling parameters of the scheduling policy attribute of a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_attr_getschedparam(**
    **const pthread_attr_t ****attr***,**
    **struct sched_param ****param***);**

## PARAMETERS

*attr* Specifies the address of the thread attributes object with the scheduling policy attribute whose scheduling parameters are obtained.

*param* Receives the values of the scheduling parameters.

## DESCRIPTION

This function obtains the values of the scheduling parameters of the scheduling policy attribute of the thread attributes object specified by the *attr* parameter and returns them in the *param* parameter.

See the **pthread_attr_setschedparam(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the scheduling parameters.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *attr* parameter is invalid. |

**RELATED INFORMATION**

Functions: **pthread_attr_init(2)**, **pthread_attr_setschedparam(2)**, **pthread_create(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_getschedpolicy -** Obtains the scheduling policy attribute of a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <pthread.h> | #include <spthread.h>
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

int pthread_attr_getschedpolicy(
        const pthread_attr_t *attr,
        int *policy);
```

## PARAMETERS

*attr* Specifies the address of the thread attributes object whose scheduling policy attribute is obtained.

*policy* Receives the value of the scheduling policy attribute.

## DESCRIPTION

This function obtains the value of the scheduling policy attribute of the thread attributes object specified by the *attr* parameter and returns it in the *policy* parameter. The scheduling policy attribute defines the scheduling policy for threads created using this threads attributes object.

See the **pthread_attr_setschedpolicy(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the scheduling policy attribute.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *attr* parameter is invalid. |

## RELATED INFORMATION

Functions: **pthread_attr_init(2)**, **pthread_attr_setschedpolicy(2)**, **pthread_create(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_getscope -** Gets the **contentionscope** attribute of a thread attributes object

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**int pthread_attr_getscope(**
    **const pthread_attr_t** ***attr*,**
    **int** ***contentionscope*);**

## PARAMETERS

*attr*
Specifies the address of the thread attributes object whose **contentionscope** attribute is obtained.

*contentionscope*
Receives the value of the **contentionscope** attribute.

## DESCRIPTION

The **pthread_attr_getscope** function obtains the value of the **contentionscope** attribute of the thread attributes object specified by the *attr* parameter and returns it in the *contentionscope* parameter.

The *contentionscope* parameter always returns the value **PTHREAD_SCOPE_PROCESS**, which signifies process scheduling contention scope. Although **PTHREAD_SCOPE_SYSTEM** and **PTHREAD_SCOPE_PROCESS** are defined in the **pthread.h** header file, only **PTHREAD_SCOPE_PROCESS** is supported.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**NOTES**

The POSIX User Thread Model library supports only the **PTHREAD_SCOPE_PROCESS** value for the *contentionscope* parameter.

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment using the POSIX User Thread Model library, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

0 Successful completion.

[EINVAL] The value specified by the *attr* parameter does not refer to an initialized thread attributes object.

**RELATED INFORMATION**

Functions: **pthread_attr_destroy(2)**, **pthread_attr_getinheritsched(2)**, **pthread_attr_getschedparam(2)**, **pthread_attr_getschedpolicy(2)**, **pthread_attr_setscope(2)**, **pthread_create(2)**.

**STANDARDS CONFORMANCE**

This function conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**pthread_attr_getstackaddr -** Obtains the stackbase address attribute of the specified thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

```
int pthread_attr_getstackaddr(
        const pthread_attr_t *attr,
        void **stackaddr);
```

## PARAMETERS

*attr* Specifies the address of the thread attributes object whose stack address attribute is obtained.

*stackaddr* Receives the value of the stack address for the thread attributes object.

## DESCRIPTION

This function obtains the value of the stackbase address attribute of the thread attributes object specified by the *attr* parameter and returns it in the *stackaddr* parameter. The specified attributes object must be initialized before this function is called.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

This function returns 0 (zero) upon successful completion of the call.

## RELATED INFORMATION

Functions: **pthread_attr_init(2)**, **pthread_getattr_np(2)**, **pthread_create(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_getstacksize -** Obtains the stacksize attribute of a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

```
int pthread_attr_getstacksize(
        const pthread_attr_t *attr,
        size_t *stacksize);
```

## PARAMETERS

*attr* Specifies the address of the thread attributes object whose **stacksize** attribute is obtained.

*stacksize* Receives the value of the **stacksize** attribute.

## DESCRIPTION

This function obtains the value of the **stacksize** attribute of the thread attributes object specified by the *attr* parameter and returns it in the *stacksize* parameter.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

### RETURN VALUES

On successful completion, this function returns a 0 (zero) and the **stacksize** attribute value is returned in *stacksize*.

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *attr* parameter is invalid. |

### RELATED INFORMATION

Functions: **pthread_attr_init(2)**, **pthread_attr_setstacksize(2)**, **pthread_create(2)**.

### STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_init -** Initializes a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_attr_init(**
    **pthread_attr_t ****attr***);**

## PARAMETERS

*attr* Specifies the address of the thread attributes object to be initialized.

## DESCRIPTION

This function initializes the thread attributes object specified by the *attr* parameter with a set of default attribute values. A thread attributes object is used to specify the attributes of threads when they are created. A thread attributes object created by this function is used only in calls to the **pthread_create( )** function.

The following functions change individual attributes of an initialized thread attributes object:

**pthread_attr_setdetachstate( )**
**pthread_attr_setguardsize_np( )**
**pthread_attr_setinheritsched( )**
**pthread_attr_setschedparam( )**
**pthread_attr_setschedpolicy( )**
**pthread_attr_setstacksize( )**

The attributes of a thread attributes object are initialized to default values. The default value of each attribute is discussed in the reference page for the corresponding function listed above.

When a thread attributes object is used to create a thread, the object's attribute values determine the characteristics of the new thread. Thus, thread attributes objects act as additional arguments to thread creation. Changing the attributes of a thread attributes object does not affect any threads that were previously created using that thread attributes object.

You can use the same thread attributes object in successive calls to **pthread_create( )**, from any thread. (However, you cannot use the same value of the stack address attribute to create multiple threads that might run concurrently; threads cannot share a stack.) If more than one thread might change the attributes in a shared thread attributes object, your program must use a mutex to protect the integrity of the thread attributes object's contents.

When you set the scheduling policy or scheduling parameters, or both, of a thread attributes object, scheduling inheritance must be disabled if you want the scheduling attributes you set to be used at thread creation. In the HP implementation, the default value of **PTHREAD_EXPLICIT_SCHED** for the *inherit* attribute of a new thread automatically disables scheduling inheritance. At thread creation, the scheduling policy and scheduling parameters stored in the thread attributes object passed to the **pthread_create( )** function are used by default. To enable scheduling inheritance, before creating the new thread use the

**pthread_attr_setinheritsched( )** function to specify the value **PTHREAD_INHERIT_SCHED** for the *inherit* parameter.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If an error condition occurs, the thread attributes object cannot be used and this function returns an integer value indicating the type of error. Possible return values are:

0 Successful completion.

[EINVAL] The value specified by the *attr* parameter is not a valid thread attributes object.

[ENOMEM] Insufficient memory exists to initialize the thread attributes object.

## RELATED INFORMATION

Functions: **pthread_attr_destroy(2)**, **pthread_attr_setdetachstate(2)**, **pthread_attr_setguardsize_np(2)**, **pthread_attr_setinheritsched(2)**, **pthread_attr_setschedparam(2)**, **pthread_attr_setschedpolicy(2)**, **pthread_attr_setstacksize(2)**, **pthread_create(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_setdetachstate -** Sets the detachstate attribute of a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_attr_setdetachstate(**
    **pthread_attr_t** ***attr*,**
    **int** *detachstate***);**

## PARAMETERS

*attr* Specifies the thread attributes object whose **detachstate** attribute is to be set.

*detachstate* Specifies the new value for the **detachstate** attribute. Valid values are:

**PTHREAD_CREATE_JOINABLE**
This is the default value. Threads are created in "undetached" state.

**PTHREAD_CREATE_DETACHED**
A created thread is detached immediately, before it begins running.

## DESCRIPTION

This function sets the value of the **detachstate** attribute of the thread attributes object specified by the *attr* parameter to the value specified by the *detachstate* parameter. The **detachstate** attribute specifies whether storage used by the thread can be reclaimed by the system when the thread terminates.

You cannot use the thread identifier (the value of type **pthread_t** that is returned by the **pthread_create( )** function) for a thread that is created detached in calls to the **pthread_detach( )** or **pthread_join( )** functions.

When a thread that has not been detached finishes executing, the system retains the state of that thread to allow another thread to join with it. If the thread is detached before it finishes executing, the system can immediately reclaim the thread's storage and resources when the thread terminates (that is, when it returns from its start routine, calls the **pthread_exit( )** function, or is canceled.)

The **pthread_join( )** or **pthread_detach( )** function should eventually be called for every thread that is created with the **detachstate** attribute of its thread attributes object set to **PTHREAD_CREATE_JOINABLE**, so that storage associated with the thread can be reclaimed.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *attr* parameter is not a valid thread attributes object or the *detachstate* parameter is invalid. |

**RELATED INFORMATION**

Functions: **pthread_attr_init(2)**, **pthread_attr_getdetachstate(2)**, **pthread_create(2)**, **pthread_join(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_setguardsize -** Sets the **guardsize** attribute of a thread attributes object

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**int pthread_attr_setguardsize(**
**pthread_attr_t ****attr***,**
**size_t** *guardsize***);**

## PARAMETERS

*attr* Specifies the address of the thread attributes object whose **guardsize** attribute is to be set.

*guardsize* Specifies the new value for the **guardsize** attribute.

## DESCRIPTION

The **pthread_attr_setguardsize** function sets the value of the **guardsize** attribute of the thread attributes object specified by the *attr* parameter to the value specified by the *guardsize* parameter.

When the protected stack feature is enabled and the guard size is 0 (zero), a regular stack is created without a guard page the next time the **pthread_create** function is called for the attributes object.

When the protected stack feature is enabled, the value of the guard size is rounded up to a multiple of a page size.

When creating a thread, use a thread attributes object to specify nondefault values for thread attributes. The **guardsize** attribute of a thread attributes object specifies the minimum size (in bytes) of the guard area for the stack of a new thread.

A guard area can help a multi-threaded program detect overflow of a thread's stack. A guard area is a region of no-access memory that the system allocates at the overflow end of the thread's stack. When any thread attempts to access a memory location within this region, a memory addressing violation occurs.

A new thread can be created with the default value for the **guardsize** attribute. This value is platform-dependent but is always at least one "hardware protection unit" (that is, at least one page).

After this function is called, the system might reserve a larger guard area for a new thread than was specified by the *guardsize* parameter.

The system allows your program to specify the size of a thread stack's guard area because:

- When a thread allocates large data structures on its stack, a guard area with a size greater than the default size might be required to detect stack overflow.
- Overflow protection of a thread's stack can potentially waste system resources, such as for an application that creates a large number of threads that will never overflow their stacks. A multi-threaded program can conserve system resources by specifying a *guardsize* parameter of 0 (zero).

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**NOTES**

This function is not supported with the Standard POSIX Threads (SPT) library. SPT-based applications should use the **pthread_attr_setguardsize_np( )** function instead.

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment using the POSIX User Thread Model library, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified for the *attr* parameter or the *guardsize* parameter is invalid. |

**RELATED INFORMATION**

Functions: **pthread_attr_init(2)**, **pthread_attr_getguardsize(2)**, **pthread_attr_setguardsize_np(2)**, **pthread_attr_setstacksize(2)**, **pthread_create(2)**.

**STANDARDS CONFORMANCE**

This function conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**pthread_attr_setguardsize_np -** Sets the **guardsize** attribute of a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

```
#include <spthread.h>

int pthread_attr_setguardsize_np(
        pthread_attr_t *attr,
        size_t guardsize );
```

## PARAMETERS

*attr* specifies the address of the thread attributes object whose **guardsize** attribute is to be set.

*guardsize* specifies the new value for the **guardsize** attribute.

## DESCRIPTION

This function sets the value of the **guardsize** attribute of the thread attributes object specified by the *attr* parameter to the value specified by the *guardsize* parameter.

When creating a thread, use a thread attributes object to specify nondefault values for thread attributes. The **guardsize** attribute of a thread attributes object specifies the minimum size (in bytes) of the guard area for the stack of a new thread.

A guard area can help a multithreaded program detect overflow of a thread's stack. A guard area is a region of no-access memory that the system allocates at the overflow end of the thread's stack. When any thread attempts to access a memory location within this region, a memory addressing violation occurs.

A new thread can be created with the default value for the **guardsize** attribute. This value is platform-dependent but is always at least one "hardware protection unit" (that is, at least one page).

After this function is called, the system might reserve a larger guard area for a new thread than was specified by the *guardsize* parameter.

The system allows your program to specify the size of a thread stack's guard area because:

- When a thread allocates large data structures on its stack, a guard area with a size greater than the default size might be required to detect stack overflow.
- Overflow protection of a thread's stack can potentially waste system resources, such as for an application that creates a large number of threads that will never overflow their stacks. A multithreaded program can conserve system resources by specifying a **guardsize** attribute of 0 (zero).

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

0 Successful completion.

[EINVAL] The value specified for the *attr* parameter or the *guardsize* parameter is invalid.

## RELATED INFORMATION

Functions: **pthread_attr_init(2)**, **pthread_attr_getguardsize_np(2)**, **pthread_attr_setstacksize(2)**, **pthread_create(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification and to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

## NAME

**pthread_attr_setinheritsched -** Sets the inherit scheduling attribute of a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <pthread.h> | #include <spthread.h>
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

int pthread_attr_setinheritsched(
        pthread_attr_t *attr,
        int inheritsched);
```

## PARAMETERS

*attr* Specifies the address of the thread attributes object whose **inherit** scheduling attribute is to be set.

*inheritsched* Specifies the new value for the **inherit** scheduling attribute. Valid values are:

**PTHREAD_INHERIT_SCHED**
The created thread inherits the scheduling policy and associated scheduling attributes of the thread calling the **pthread_create( )** function. Any scheduling attributes in the thread attributes object specified by the **pthread_create( )** *attr* parameter are ignored during thread creation.

**PTHREAD_EXPLICIT_SCHED**
This is the default value. The scheduling policy and associated scheduling attributes of the created thread are set to the corresponding values from the thread attributes object specified by the **pthread_create( )** *attr* parameter.

## DESCRIPTION

This function sets the value of the **inherit** scheduling attribute of the thread attributes object specified by the *attr* parameter to the value specified by the *inheritsched* parameter. The **inherit** scheduling attribute specifies whether threads created using the specified thread attributes object inherit the scheduling attributes of the creating thread or use the scheduling attributes stored in the thread attributes object specified by the **pthread_create( )** *attr* parameter.

The default scheduling policy for the first thread in an application is **SCHED_FIFO**, and cannot be modified.

Inheriting scheduling attributes is useful when a thread is creating several helper threads — that is, threads that are intended to work closely with the creating thread to cooperatively solve the same problem. For example, inherited scheduling attributes ensure that helper threads created in a sort routine execute with the same priority as the calling thread.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *attr* parameter is not a valid thread attributes object or the *inheritsched* parameter contains an invalid value. |
| [ENOTSUP] | An attempt was made to set the attribute to an unsupported value. |

## RELATED INFORMATION

Functions: **pthread_attr_init(2)**, **pthread_attr_getinheritsched(2)**, **pthread_attr_setschedpolicy(2)**, **pthread_attr_setschedparam(2)**, **pthread_create(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_setschedparam -** Sets the scheduling parameters of the scheduling policy attribute of a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_attr_setschedparam(**
    **pthread_attr_t ****attr***,**
    **const struct sched_param ****param***);**

## PARAMETERS

*attr* Specifies the address of the thread attributes object with the scheduling policy attribute whose scheduling parameters are to be set.

*param* Specifies a structure containing the new values for the scheduling parameters.

The system provides only the **sched_priority** scheduling parameter. See **Description** for information about this scheduling parameter.

## DESCRIPTION

This function sets the values of the scheduling parameters of the scheduling policy attribute of the thread attributes object specified by the *attr* parameter to the values specified by the *param* parameter.

Use the **sched_priority** field of the **sched_param** structure to set a thread's execution priority. The effect of the scheduling priority you assign depends on the scheduling policy attribute of the thread attributes object specified by the *attr* parameter.

By default, the prioirty of a created thread is determined by the *attr* parameter used in the call to the **pthread_create( )** function. To inherit the prioirty of the thread calling **pthread_create( )**, scheduling inheritance must be enabled when the thread is created. Before calling **pthread_create( )**, call **pthread_attr_setinheritsched( )** and specify the value **PTHREAD_INHERIT_SCHED** for the *inherit* parameter.

An application specifies priority only to express the urgency of executing the thread relative to other threads. DO NOT USE PRIORITY TO CONTROL MUTUAL EXCLUSION WHEN ACCESSING SHARED DATA. With a sufficient number of processors present, all ready threads, regardless of priority, execute simultaneously.

Valid values of the **sched_priority** scheduling parameter depend on the chosen scheduling policy. Use the **sched_get_priority_min( )** and **sched_get_priority_max( )** functions to determine the low and high limits of each policy.

Open System Services provides the following nonportable priority range constants:

**SCHED_FIFO**
**PRI_FIFO_MIN** to **PRI_FIFO_MAX**

**SCHED_RR** **PRI_RR_MIN** to **PRI_RR_MAX**

**SCHED_OTHER**
**PRI_OTHER_MIN** to **PRI_OTHER_MAX**

**SCHED_FG_NP**
**PRI_FG_MIN_NP** to **PRI_FG_MAX_NP**

**SCHED_BG_NP**
**PRI_BG_MIN_NP** to **PRI_BG_MAX_NP**

The default priority is 24.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values ares:

0 Successful completion.

[EINVAL] The value specified by the *attr* parameter is not a valid thread attributes object or the value specified by *param* is invalid.

[ENOTSUP] An attempt was made to set the attribute to an unsupported value.

**RELATED INFORMATION**

Functions: **pthread_attr_init(2)**, **pthread_attr_getschedparam(2)**, **pthread_attr_setinheritsched(2)**, **pthread_attr_setschedpolicy(2)**, **pthread_create(2)**, **sched_yield(2)**, **sched_get_priority_max(2)**, **sched_get_priority_min(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_setschedpolicy** - Sets the scheduling policy attribute of a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_attr_setschedpolicy(**
    **pthread_attr_t ****attr***,**
    **int** *policy***);**

## PARAMETERS

*attr* Specifies the address of the thread attributes object whose scheduling policy attribute is to be set.

*policy* Specifies the new value for the scheduling policy attribute. Valid values are:

**SCHED_FIFO** is the default value and the only value supported.

## DESCRIPTION

This function sets the value of the scheduling policy attribute of the thread attributes object specified by the *attr* parameter to the value specified by the *policy* attribute. The only supported policy is **SCHED_FIFO**. An attempt to change this value returns the value of [ENOTSUP] for this function.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**NOTES**

Never attempt to use scheduling as a mechanism for synchronization.

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *policy* parameter is invalid. |
| [ENOTSUP] | An attempt was made to set the scheduling policy to an unsupported value. |

**RELATED INFORMATION**

Functions: **pthread_attr_init(2)**, **pthread_attr_getschedpolicy(2)**, **pthread_attr_setinheritsched(2)**, **pthread_attr_setschedparam(2)**, **pthread_create(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_attr_setscope -** Sets the **contentionscope** attribute of a thread attributes object

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setscope(
        pthread_attr_t *attr,
        int contentionscope);
```

## PARAMETERS

*attr* Specifies the address of the thread attributes object whose **contentionscope** attribute is to be set.

*contentionscope*
Specifies the value to which the **contentionscope** attribute is to be set.

## DESCRIPTION

The **pthread_attr_setscope** function sets the value of the contentionscope attribute of the thread attributes object specified by the *attr* parameter to the value specified in the *contentionscope* parameter.

The only valid value supported for the *contentionscope* parameter is **PTHREAD_SCOPE_PROCESS**, which signifies process scheduling contention scope. **PTHREAD_SCOPE_SYSTEM** and **PTHREAD_SCOPE_PROCESS** are defined in the **pthread.h** header file.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**NOTES**

The POSIX User Thread Model library supports only the **PTHREAD_SCOPE_PROCESS** value for the *contentionscope* parameter.

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment using the POSIX User Thread Model library, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *attr* parameter is not valid, or the value specified by the *contentionscope* parameter is not valid. |
| [ENOTSUP] | The value specified by the *contentionscope* parameter is **PTHREAD_SCOPE_SYSTEM**. **PTHREAD_SCOPE_SYSTEM** is not a supported value. |

**RELATED INFORMATION**

Functions: **pthread_attr_destroy(2)**, **pthread_attr_getinheritsched(2)**, **pthread_attr_getschedparam(2)**, **pthread_attr_getschedpolicy(2)**, **pthread_attr_getscope(2)**, **pthread_create(2)**.

**STANDARDS CONFORMANCE**

This function conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**pthread_attr_setstacksize -** Sets the stacksize attribute of a thread attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

```
int pthread_attr_setstacksize(
        pthread_attr_t *attr,
        size_t stacksize);
```

## PARAMETERS

*attr* Specifies the address of the thread attributes object whose **stacksize** attribute is to be set.

*stacksize* Specifies the new value for the **stacksize** attribute. The *stacksize* parameter must be greater than or equal to **PTHREAD_STACK_MIN**, which is the minimum size (in bytes) of stack needed for a thread.

## DESCRIPTION

This function sets the value of the **stacksize** attribute in the thread attributes object specified by the *attr* parameter to the value specified by the *stacksize* parameter. Use this function to adjust the size of the writable area of the stack for a new thread.

The size of a thread's stack is fixed at the time of thread creation. Only the initial thread can dynamically extend its stack.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**NOTES**

Many compilers do not check for stack overflow. Ensure that the new thread's stack is big enough for the resources required by routines that are called from the thread.

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *attr* parameter is invalid, or the value specified by the *stacksize* parameter either is less than **PTHREAD_STACK_MIN** or exceeds a system-imposed limit. |

**RELATED INFORMATION**

Functions: **pthread_attr_init(2)**, **pthread_attr_getstacksize(2)**, **pthread_create(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_cancel -** Requests that a thread terminate execution

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_cancel(**
    **pthread_t** *thread***);**

## PARAMETERS

*thread* Specifies the thread that receives the cancelation request.

## DESCRIPTION

This function sends a cancelation request to the specified target thread. A cancelation request is a mechanism by which a calling thread requests the target thread to terminate as quickly as possible. Issuing a cancelation request does not guarantee that the target thread receives or handles the request.

When the cancelation request is acted on, all active cleanup-handler routines for the target thread are called. When the last cleanup handler returns, the thread-specific data destructor routines are called for each thread-specific data key with a destructor and for which the target thread has a non-**NULL** value. Finally, the target thread is terminated, and a status of **PTHREAD_CANCELED** is made available to any threads joining with the target thread.

Cancelation of the target thread runs asynchronously to the calling thread's returning from **pthread_cancel( )**. The target thread's cancelability state and type determine when or if the cancelation takes place, as follows:

- The target thread can delay cancelation during critical operations by setting its cancelability state to **PTHREAD_CANCEL_DISABLE**.
- Because of communication delays, the calling thread can rely only on the fact that a cancelation request eventually becomes pending in the target thread (provided that the target thread does not terminate beforehand).
- The calling thread has no guarantee that a pending cancelation request will be delivered, because delivery is controlled by the target thread.

When a cancelation request is delivered to a thread, termination processing is similar to that for **pthread_exit( )**. For more information about thread termination, see the **pthread_create(2)** reference page either online or in the *Open System Services System Calls Reference Manual*.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the

following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [ESRCH] | The value of the *thread* parameter does not specify an existing thread. |

## RELATED INFORMATION

Functions: **pthread_cleanup_pop(2)**, **pthread_cleanup_push(2)**, **pthread_create(2)**, **pthread_exit(2)**, **pthread_join(2)**, **pthread_setcancelstate(2)**, **pthread_setcanceltype(2)**, **pthread_testcancel(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_cleanup_pop -** (Macro) Removes the cleanup-handler routine from the calling thread's cleanup-handler stack and optionally executes it

## LIBRARY

None. This application program interface is implemented as a macro.

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**void pthread_cleanup_pop(**
  **int** *execute***);**

## PARAMETERS

*execute* Controls whether the cleanup-handler routine specified in the matching call to **pthread_cleanup_push( )** is executed. If *execute* is nonzero, the cleanup-handler routine executes.

## DESCRIPTION

This macro removes the cleanup-handler routine established by the matching call to **pthread_cleanup_push( )** from the calling thread's cleanup-handler stack, then executes it if the value of *execute* is nonzero.

A cleanup-handler routine can be used to clean up from a block of code whether the code is exited by normal completion, cancelation, or the raising (or reraising) of an exception. The routine is popped from the calling thread's cleanup-handler stack and is executed with its *arg* parameter when any of the following actions occur:

- The thread calls **pthread_cleanup_pop( )** and specifies a nonzero value for the *execute* parameter.
- The thread calls **pthread_exit( )**.
- The thread is canceled.
- An exception is raised and is caught when the system unwinds the calling thread's stack to the lexical scope of the **pthread_cleanup_push( )** and **pthread_cleanup_pop( )** macros.

This macro and **pthread_cleanup_push( )** must appear in pairs within the same lexical scope.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this macro in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

To use this macro in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.

## RELATED INFORMATION

Functions: **pthread_cancel(2)**, **pthread_cleanup_push(2)**, **pthread_create(2)**, **pthread_exit(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_cleanup_push -** (Macro) Establishes a cleanup-handler routine to be executed when the thread terminates

## LIBRARY

None. This application program interface is implemented as a macro.

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**void pthread_cleanup_push(**
**void (*_routine_) (void *),**
**void *_arg_);**

## PARAMETERS

*routine* Specifies the routine to be executed as the cleanup handler.

*arg* Specifies an argument to be passed to the cleanup routine.

## DESCRIPTION

This macro pushes the specified routine onto the calling thread's cleanup- handler stack. The cleanup-handler routine is popped from the stack and executed with the value specified by the *arg* parameter when any of the following actions occur:

- The thread calls **pthread_cleanup_pop( )** and specifies a nonzero value for the *execute* parameter.
- The thread calls **pthread_exit( )**.
- The thread is canceled.
- An exception is raised and is caught when the system unwinds the calling thread's stack to the lexical scope of the **pthread_cleanup_push( )** and **pthread_cleanup_pop( )** pair.

This routine and **pthread_cleanup_pop( )** must appear in pairs within the same lexical scope.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this macro in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

To use this macro in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.

## RELATED INFORMATION

Functions: **pthread_cancel(2)**, **pthread_cleanup_pop(2)**, **pthread_create(2)**, **pthread_exit(2)**, **pthread_testcancel(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_condattr_destroy -** Destroys a condition variable attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_condattr_destroy(**
    **pthread_condattr_t ****attr***);**

## PARAMETERS

*attr* Specifies the condition variable attributes object to be destroyed.

## DESCRIPTION

This function destroys the specified condition variable attributes object by uninitializing the object.

Destroying an attributes object does not affect any condition variables that were created using that attributes object.

After this function is called, using the value of *attr* in a call to any function other than the **pthread_condattr_init( )** function returns an error.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## NOTES

The **pthread_condattr_init( )** and **pthread_condattr_destroy( )** functions are provided for future expansion of the threads interface and to conform with the POSIX.1c standard. These functions are not currently useful, because the functions to set and get the process shared attribute are not supported by this implementation.

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible error return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The condition variable attributes object specified by the *attr* parameter is invalid. |

## RELATED INFORMATION

Functions: **pthread_condattr_init(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_condattr_init -** Initializes a condition variable attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_condattr_init(**
    **pthread_condattr_t ***attr***);**

## PARAMETERS

*attr* Specifies the condition variable attributes object to be initialized.

For the Standard POSIX Threads library, if the value specified is **pthread_condattr_default**, then the default attribute is:

**PTHREAD_PROCESS_PRIVATE**
Specifies that the initialized condition variable can be used only within a process.

## DESCRIPTION

This function initializes the condition variable attributes object specified by the *attr* parameter with a set of default attribute values.

When an attributes object is used to create a condition variable, the values of the individual attributes determine the characteristics of the new condition variable. Attributes objects act as additional arguments to creation of condition variables. Changing individual attributes in an attributes object does not affect any condition variables that were previously created using that attributes object.

You can use the same condition variable attributes object in successive calls to **pthread_condattr_init( )** from any thread. If multiple threads can change attributes in a shared condition variable attributes object, your program must use a mutex to protect the integrity of the contents of that condition variable attributes object.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## NOTES

The **pthread_condattr_init( )** and **pthread_condattr_destroy( )** functions are provided for future expansion of the threads interface and to conform with the POSIX.1c standard. These functions are not currently useful because the functions to set and get the process share attribute are not supported by this implementation.

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [ENOMEM] | Insufficient memory exists to initialize the condition variable attributes object. |

## RELATED INFORMATION

Functions: **pthread_cond_init(2)**, **pthread_condattr_destroy(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_cond_broadcast -** Unblocks all threads that are waiting on the specified condition variable

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_cond_broadcast(**
**pthread_cond_t ***cond***);**

## PARAMETERS

*cond* Specifies a condition variable upon which the threads (to be awakened) are waiting.

## DESCRIPTION

This function unblocks all threads waiting on the condition variable specified by *cond*. Calling this function implies that data guarded by the associated mutex has changed, so one or more waiting threads might be able to proceed. The threads that are unblocked contend for the mutex according to their respective scheduling policies (if applicable).

This function can be called by a thread regardless of whether it currently owns the mutex associated with the condition variable specified by *cond*. However, if predictable scheduling behavior is required, the mutex must be locked before the **pthread_cond_broadcast( )** function is called.

If no threads are waiting on the specified condition variable, this function takes no action. The broadcast does not propagate to the next condition variable wait.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *cond* parameter is invalid. |
| [ENOMEM] | There is insufficient memory to initialize the condition variable specified by the *cond* parameter. |

**RELATED INFORMATION**

Functions: **pthread_cond_destroy(2)**, **pthread_cond_init(2)**, **pthread_cond_signal(2)**, **pthread_cond_timedwait(2)**, **pthread_cond_wait(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

The return of [ENOMEM] is an HP extension to the POSIX standard.

## NAME

**pthread_cond_destroy -** Destroys a condition variable

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <pthread.h> | #include <spthread.h>
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

int pthread_cond_destroy(
        pthread_cond_t *cond);
```

## PARAMETERS

*cond* Specifies the condition variable to be destroyed.

## DESCRIPTION

This function destroys the condition variable specified by the *cond* parameter. This function effectively uninitializes the condition variable. Call this function when a condition variable will no longer be referenced. Destroying a condition variable allows the system to reclaim internal memory associated with the condition variable.

It is safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are blocked results in an error and returns the value of [EBUSY].

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EBUSY] | The object referenced by *cond* is being referenced by another thread that is currently executing **pthread_cond_wait( )** or **pthread_cond_timedwait( )** on the condition variable specified in *cond*. |
| [EINVAL] | The value specified by the *cond* parameter is invalid. |

**RELATED INFORMATION**

Functions: **pthread_cond_broadcast(2)**, **pthread_cond_init(2)**, **pthread_cond_signal(2)**, **pthread_cond_timedwait(2)**, **pthread_cond_wait(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_cond_init -** Initializes a condition variable

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_cond_init(**
    **pthread_cond_t ****cond***,**
    **const pthread_condattr_t ****attr***);**

## PARAMETERS

*cond* Specifies the condition variable to be initialized.

*attr* Specifies the condition variable attributes object that defines the characteristics of the condition variable to be initialized.

## DESCRIPTION

This function initializes the condition variable specified by the *cond* parameter with attributes indicated by the *attr* parameter. If the value of *attr* is **NULL**, the default condition variable attributes are used.

A condition variable is a synchronization object used with a mutex. A mutex controls access to data that is shared among threads; a condition variable allows threads to wait for that data to enter a defined state.

Condition variables are not owned by a particular thread. Any associated storage is not automatically deallocated when the creating thread terminates.

If the default condition variable attributes are appropriate, use the macro **PTHREAD_COND_INITIALIZER** to initialize statically allocated condition variables. The effect of using this macro is the same as the effect of calling **pthread_cond_init( )** with an *attr* parameter of **NULL**. To call this macro, specify:

**pthread_cond_t condition = PTHREAD_COND_INITIALIZER**;

When statically initialized, a condition variable should not also be used in the **pthread_cond_init( )** function.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error, the condition variable is not initialized, and the contents of *cond* are undefined. Possible return values are:

0 Successful completion.

[EAGAIN] One of the following conditions exists:

- The system lacks the necessary resources to initialize another condition variable.
- The system-imposed limit on the total number of condition variables under execution by a single user is exceeded.

[EBUSY] The implementation has detected an attempt to reinitialize the object indicated by *cond*, a previously initialized, but not yet destroyed, condition variable.

[EINVAL] The value specified by the *attr* parameter is invalid.

[ENOMEM] Insufficient memory exists to initialize the condition variable.

**RELATED INFORMATION**

Functions: **pthread_cond_broadcast(2)**, **pthread_cond_destroy(2)**, **pthread_cond_signal(2)**, **pthread_cond_timedwait(2)**, **pthread_cond_wait(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_cond_signal -** Unblocks at least one thread that is waiting on the specified condition variable

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_cond_signal(**
    **pthread_cond_t ***cond***);**

## PARAMETERS

*cond* Specifies the condition variable to be signaled.

## DESCRIPTION

This function unblocks at least one thread waiting on the condition variable specified by *cond*. Calling this function implies that data guarded by the associated mutex has changed, so one of the waiting threads might be able to proceed. In general, only one thread is unblocked.

If no threads are waiting on the specified condition variable, this function takes no action. The signal does not propagate to the next condition variable wait.

The scheduling policy determines which thread is unblocked. A blocked thread is chosen in priority order, using a first-in/first-out (FIFO) algorithm within priorities.

This function can be called by a thread regardless of whether it owns the mutex associated with the condition variable specified by the *cond* parameter. However, if predictable scheduling behavior is required, the mutex must be locked before the **pthread_cond_signal( )** function is called.

Do not call this function from within an interrupt handler.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *cond* parameter is not a valid condition variable. |
| [ENOMEM] | There is insufficient memory to perform the requested operation. |

## RELATED INFORMATION

Functions: **pthread_cond_broadcast(2)**, **pthread_cond_destroy(2)**, **pthread_cond_init(2)**, **pthread_cond_timedwait(2)**, **pthread_cond_wait(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

The return of [ENOMEM] is an HP extension to the POSIX standard.

## NAME

**pthread_cond_signal_int_np -** Unblocks one thread that is waiting on the specified condition variable; callable only from an interrupt-handler routine

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <pthread.h> | #include <spthread.h>
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

int pthread_cond_signal_int_np(
        pthread_cond_t *cond);
```

## PARAMETERS

*cond* Specifies the condition variable to be signaled.

## DESCRIPTION

This function unblocks one thread waiting on the condition variable specified by *cond*. Calling this function implies that data guarded by the associated mutex has changed, so the waiting thread might be able to proceed.

If no threads are waiting on the specified condition variable, this function takes no action. The signal does not propagate to the next condition variable wait.

The scheduling policy of the waiting threads determines which thread is unblocked. A blocked thread is chosen in priority order, using a first-in/first-out (FIFO) algorithm within priorities.

This function does not cause a thread blocked on a condition variable to resume execution immediately. The thread resumes execution at some time after the interrupt-handler routine returns.

You can call this function regardless of whether the associated mutex is locked by some other thread. Never lock a mutex from an interrupt- handler routine.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## NOTES

This function allows you to signal a thread from a software interrupt handler. Do not call this function from noninterrupt code. To signal a thread from the normal noninterrupt level, use the **pthread_cond_signal( )** function.

## RETURN VALUES

On successful completion, this function returns a 0 (zero). If an error condition occurs, this function returns -1 and sets **errno** to indicate the type of error.

## ERRORS

The following error conditions can occur:

[EINVAL] The value specified by the *cond* parameter is not a valid condition variable.

## RELATED INFORMATION

Functions: **pthread_cond_broadcast(2)**, **pthread_cond_destroy(2)**, **pthread_cond_init(2)**, **pthread_cond_timedwait(2)**, **pthread_cond_wait(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification and to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

**NAME**

**pthread_cond_timedwait -** Causes a thread to wait either for a condition variable to be signaled or broadcast, or for a specific expiration time

**LIBRARY**

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

**SYNOPSIS**

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_cond_timedwait(**
    **pthread_cond_t** *****cond***,**
    **pthread_mutex_t** *****mutex***,**
    **const struct timespec** *****abstime***);**

**PARAMETERS**

*cond* Specifies the condition variable that the calling thread waits on.

*mutex* Specifies the mutex associated with the condition variable specified by the *cond* parameter.

*abstime* Specifies the absolute time at which the wait expires, if the condition variable *cond* has not been signaled or broadcast. See the **pthread_get_expiration_np(2)** reference page either online or in the *Open System Services System Calls Reference Manual*; that function is used to obtain a value for this parameter. The *abstime* value is specified in Universal Coordinated Time (UTC).

**DESCRIPTION**

This function causes a thread to wait until one of the following occurs:

- The specified condition variable is signaled or broadcasted.
- The current system clock time is greater than or equal to the time specified by the *abstime* parameter.

This function is similar to the **pthread_cond_wait( )** function, except that this function can return before a condition variable is signaled or broadcast if the specified time expires. For more information, see the **pthread_cond_wait(2)** reference page either online or in the *Open System Services System Calls Reference Manual*.

This function atomically releases the mutex and causes the calling thread to wait on the condition variable. When the thread regains control after calling **pthread_cond_timedwait( )**, the mutex is locked and the thread is the owner, regardless of why the wait ended. If general cancelability is enabled, the thread reacquires the mutex (blocking for it if necessary) before the cleanup handlers are run (or before the exception is raised).

If the current time equals or exceeds the expiration time, this function returns immediately, releasing and reacquiring the mutex. This function might cause the calling thread to yield (see the **sched_yield(2)** reference page either online or in the *Open System Services System Calls*

*Reference Manual*). Your code should check the return status whenever this function returns and take the appropriate action. Otherwise, waiting on the condition variable can become a nonblocking loop.

Call this function after you have locked the mutex specified by *mutex*. The results of this function are unpredictable if this function is called before the mutex is locked.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |

[EINVAL] One of the following conditions exists:

- The value specified by *cond*, *mutex*, or *abstime* is invalid.
- Different mutexes are supplied for concurrent **pthread_cond_timedwait( )** operations or **pthread_cond_wait( )** operations on the same condition variable.
- The mutex was not owned by the calling thread at the time of the call.

[ENOMEM] The system cannot acquire the memory needed to block using a statically initialized condition variable.

[ETIMEDOUT]
The time specified by the *abstime* parameter expired.

## RELATED INFORMATION

Functions: **pthread_cond_broadcast(2)**, **pthread_cond_destroy(2)**, **pthread_cond_init(2)**, **pthread_cond_signal(2)**, **pthread_cond_wait(2)**, **pthread_get_expiration_np(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

The return of [ENOMEM] is an HP extension to the POSIX standard.

## NAME

**pthread_cond_wait -** Causes a thread to wait for the specified condition variable to be signaled or broadcast

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_cond_wait(**
    **pthread_cond_t ***cond***,**
    **pthread_mutex_t ***mutex***);**

## PARAMETERS

*cond* Specifies the condition variable that the calling thread waits on.

*mutex* Specifies the mutex associated with the condition variable specified by the *cond* parameter.

## DESCRIPTION

This function causes a thread to wait for the specified condition variable to be signaled or broadcast. Each condition corresponds to one or more Boolean relations, called a predicate, based on shared data. The calling thread waits for the data to reach a particular state for the predicate to become true. However, return from this function does not imply anything about the value of the predicate, and it should be reevaluated upon return.

This function atomically releases the mutex and causes the calling thread to wait on the condition variable. When the thread regains control after calling **pthread_cond_wait( )**, the mutex is locked and the thread is the owner, regardless of why the wait ended. If general cancelability is enabled, the thread reacquires the mutex (blocking for it if necessary) before the cleanup handlers are run (or before the exception is raised).

If a thread changes the state of storage protected by the mutex in such a way that a predicate associated with a condition variable might now be true, that thread must call either the **pthread_cond_signal( )** or **pthread_cond_broadcast( )** function for that condition variable. If neither call is made, any thread waiting on the condition variable continues to wait.

This function might (with low probability) return when the condition variable has not been signaled or broadcast. When this occurs, the mutex is reacquired before the function returns. To handle this type of situation, enclose each call to this function in a loop that checks the predicate. The loop documents your intent and protects against spurious wakeups while allowing correct behavior even if another thread consumes the desired state before the awakened thread runs.

Threads are not allowed to wait on the same condition variable by specifying different mutexes.

Call this function after you have locked the mutex specified by *mutex*. The results of this function are unpredictable if this function is called before the mutex is locked.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

0 Successful completion.

[EINVAL] One of the following conditions exists:

- The value specified by the *cond* or *mutex* parameter is invalid.
- Different mutexes are supplied for concurrent **pthread_cond_wait( )** operations or **pthread_cond_timedwait( )** operations on the same condition variable.

- The mutex was not owned by the calling thread at the time of the call.

[ENOMEM] The system cannot acquire the memory needed to block using a statically initialized condition variable.

**RELATED INFORMATION**

Functions: **pthread_cond_broadcast(2)**, **pthread_cond_destroy(2)**, **pthread_cond_init(2)**, **pthread_cond_signal(2)**, **pthread_cond_timedwait(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

The return of [ENOMEM] is an HP extension to the POSIX standard.

## NAME

**pthread_create -** Creates a thread

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_create(**
    **pthread_t ***thread**,**
    **const pthread_attr_t ***attr**,**
    **void * (***start_routine**) (void *),**
    **void ***arg**);**

## PARAMETERS

*thread* Specifies the location to receive the identifier for the thread being created.

*attr* Specifies the thread attributes object that defines the characteristics of the thread being created. If you specify **NULL**, then the default attributes are:

**SCHED_FIFO**
The default scheduling policy for the **schedpolicy** attribute is first-in, first-out.

**PTHREAD_CREATE_JOINABLE**
The default **detachstate** is joinable.

For the Standard POSIX Threads library, you can also specify **pthread_attr_default** to set these default attributes.

A thread is detached when created if the **detachstate** attribute of its thread object is set to **PTHREAD_CREATE_DETACHED**.

*start_routine* Specifies the function to be executed as the new thread's start routine.

*arg* Specifies the argument to the thread's start routine.

## DESCRIPTION

This function creates a thread, which is a single, sequential flow of control within a program. A thread is the active execution of a designated routine, including any nested routine invocations.

Successful execution of this function causes the following actions:

- The system creates a thread object to describe and control the thread.
- The *thread* parameter receives an identifier for the new thread.

- An executable thread is created with attributes specified by the *attr* parameter (or with default attributes if *attr* is **NULL**).

**Thread Creation**

The system creates a thread in the ready state and prepares the thread to begin executing its start routine, which is the function passed to **pthread_create( )** as the *start_routine* parameter. Depending on the presence of other threads and their scheduling and priority attributes, the new thread might start executing immediately. The new thread can also preempt its creator, depending on the two threads' respective scheduling and priority attributes. The caller of **pthread_create( )** can synchronize with the new thread using either the **pthread_join( )** function or any mutually agreed upon mutexes or condition variables.

For the duration of the new thread's existence, the system maintains and manages the thread object and other thread state overhead.

The system assigns each new thread a thread identifier, which the system writes into the address specified by the *thread* parameter before the new thread executes.

At thread creation, the scheduling policy and scheduling parameters stored in the thread attributes object passed to **pthread_create( )** is used by default. If you want the scheduling attributes to be inherited from the parent thread, then before creating the new thread your program must use the **pthread_attr_setinheritsched( )** function with the *inheritsched* parameter set to **PTHREAD_INHERIT_SCHED**.

The signal state of the new thread is initialized as follows:

- The signal mask is inherited from the creating thread.
- The set of signals pending for the new thread is empty.

If **pthread_create( )** fails, no new thread is created and the contents of the location indicated by the *thread* parameter are undefined.

**Thread Termination**

A thread terminates when one of the following occurs:

- The thread returns from its start routine.
- The thread calls the **pthread_exit( )** function.
- The thread is canceled.

When a thread terminates, the system performs these actions:

- The system writes a return value (if one is available) into the terminated thread's attributes object, as follows:

  — If the thread has been canceled, the system writes the value **PTHREAD_CANCELED** into the object specified by *attr*.

  — If the thread terminated by returning from its start routine, the system copies the return value from the start routine (if one is available) into the object specified by *attr*.

  — If the thread explictly called the **pthread_exit( )** function, the system stores the value received in the *value_ptr* parameter of **pthread_exit( )** into the object specified by *attr*.

  Another thread can obtain this return value by joining with the terminated thread using the **pthread_join( )** function. If the thread terminated by returning from its start routine normally and the start routine does not provide a return value, the results obtained by joining with that thread are unpredictable.

- If the termination results from a cancelation request or a call to **pthread_exit( )**, the system calls, in turn, each cleanup handler that this thread declared using the **pthread_cleanup_push( )** macro that has not yet been removed using the **pthread_cleanup_pop( )** macro. (The system also transfers control to any appropriate **CATCH**, **CATCH_ALL**, or **FINALLY** blocks.)

  For C++: At normal exit from a thread, a program calls the appropriate destructor functions, just as if an exception had been raised.

- To exit a thread terminated by a call to **pthread_exit( )**, the system raises the **pthread_exit_e** exception. To exit a thread terminated by cancelation, the system raises the **pthread_cancel_e** exception. Your program can use the exception package to operate on the generated exception.

- For each of the terminated thread's thread-specific data keys that has a non-**NULL** value and a non-**NULL** destructor pointer, the system sets the thread's value for the corresponding key to **NULL**.

  In turn, the system calls each thread-specific data destructor function in this multithreaded process's list of destructors. The destructor is given the value previously associated with the data key as its sole argument. The destructor must delete all storage associated with the data key; otherwise, the destructor will be called again.

  The system repeats this step either until all thread-specific data values in the thread are **NULL** or for up to a number of iterations equal to **PTHREAD_DESTRUCTOR_ITERATIONS**. This action should destroy all thread-specific data associated with the terminated thread.

- The system unblocks the thread (if there is one) that is currently waiting to join with the terminated thread. That is, the system unblocks the thread that is waiting in a call to **pthread_join( )**.

- If the thread is already detached, the system destroys its thread object. Otherwise, the thread continues to exist until it is detached or joined with.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## NOTES

The practice of using **CATCH** handlers in place of **pthread_cleanup_push( )** is not portable.

## RETURN VALUES

If an error condition occurs, no thread is created, the contents of *thread* are undefined, and this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EAGAIN] | The system lacks the necessary resources to create another thread, or the system-imposed limit on the total number of threads under execution by a single user is exceeded. |
| [EINVAL] | The value specified by the *attr* parameter is invalid. |

## RELATED INFORMATION

Functions: **pthread_atfork(2)**, **pthread_attr_destroy(2)**, **pthread_attr_init(2)**, **pthread_attr_setdetachstate(2)**, **pthread_attr_setinheritsched(2)**, **pthread_attr_setschedparam(2)**, **pthread_attr_setschedpolicy(2)**, **pthread_attr_setstacksize(2)**, **pthread_cancel(2)**, **pthread_cleanup_pop(2)**, **pthread_cleanup_push(2)**, **pthread_detach(2)**, **pthread_exit(2)**, **pthread_join(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_delay_np -** Delays execution of a thread

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_delay_np(**
**const struct timespec ****interval***);**

## PARAMETERS

*interval* Specifies the number of seconds and nanoseconds to delay execution. The value specified for each must be greater than or equal to 0 (zero).

## DESCRIPTION

This function causes a thread to delay execution for a specific interval of time. This interval ends at the current time plus the specified interval. The function does not return before the end of the interval is reached, but it might return an arbitrary amount of time after the end of the interval is reached, because of system load, thread priorities, and system timer granularity.

Specifying an interval of 0 (zero) seconds and 0 (zero) nanoseconds is allowed and can be used to force the thread to give up the processor or to deliver a pending cancelation request.

The **timespec** structure contains the following two fields:

**tv_sec** Is an integral number of seconds.

**tv_nsec** Is an integral number of nanoseconds.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running

H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *interval* parameter is invalid. |

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification and to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

## NAME

**pthread_detach -** Marks a thread object for deletion

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_detach(**
    **pthread_t** *thread***);**

## PARAMETERS

*thread* Specifies the thread object being marked for deletion.

## DESCRIPTION

This function marks the specified thread object to indicate that storage for the corresponding thread can be reclaimed when the thread terminates. This storage includes storage for the *thread* parameter's return value, as well as for the thread object. If the specified thread has not terminated when this function is called, this function does not cause it to terminate.

A thread can be created already marked for deletion by setting its thread object's **detachstate** attribute using the **pthread_attr_setdetachstate( )** function before creating the thread.

Once detached, the use of the thread's thread identifier in a call to the **pthread_join( )** function results in an error. A joinable thread is implicitly detached when **pthread_join( )** is called.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**NOTES**

The results of this function are unpredictable if the *thread* parameter refers to a thread object that does not exist.

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The *thread* parameter does not specify a joinable thread. |
| [ESRCH] | The object specified by the *thread* parameter cannot be found. |

**RELATED INFORMATION**

Functions: **pthread_attr_getdetachstate(2)**, **pthread_attr_setdetachstate(2)**, **pthread_cancel(2)**, **pthread_create(2)**, **pthread_exit(2)**, **pthread_join(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_equal -** Compares two thread identifiers

## LIBRARY

None. This routine has been implemented as a macro.

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_equal(**
    **pthread_t** *t1***,**
    **pthread_t** *t2***);**

## PARAMETERS

*t1* Specifies the first thread identifier to be compared.

*t2* Specifies the second thread identifier to be compared.

## DESCRIPTION

This macro compares two thread identifiers.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this macro in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

To use this macro in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.

## NOTES

If either *t1* or *t2* is not a valid thread identifier, this macro's behavior is undefined.

## RETURN VALUES

Possible return values are:

0 The *t1* and *t2* parameters do not designate the same object.

Nonzero The *t1* and *t2* parameters designate the same object.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_exit -** Terminates the calling thread

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**void pthread_exit(**
**void ****value_ptr*****);**

## PARAMETERS

*value_ptr* Specifies the value to be copied and returned to the caller of the **pthread_join( )** function. Note that **void *** is used as a universal datatype, not as a pointer. The system treats the *value_ptr* parameter as a value and stores it to be returned by **pthread_join( )**.

## DESCRIPTION

This function terminates the calling thread and makes a status value (*value_ptr*) available to any thread that calls **pthread_join( )** to join the terminating thread.

Any cleanup handlers that have been pushed and not yet popped from the stack are popped in the reverse order that they were pushed and then executed. After all cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions are called. Thread termination does not release any application-visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process-level cleanup actions, including, but not limited to, calling any **atexit** routine that might exist.

An implicit call to **pthread_exit( )** is issued when a thread returns from the start routine that was used to create it. The system writes the function's return value as the return value in the thread's thread object. The process exits with an exit status of 0 (zero) when the last running thread calls **pthread_exit( )**.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## NOTES

After a thread has terminated, the result of access to local (that is, explicitly or implicitly declared **auto**) variables of the thread is undefined. References to local variables of the existing thread should not be used for the *value_ptr* parameter of the **pthread_exit( )** function.

## RELATED INFORMATION

Functions: **pthread_cancel(2)**, **pthread_create(2)**, **pthread_detach(2)**, **pthread_join(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_getattr_np -** Gets the attribute object for a thread

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_getattr_np(**
    **const pthread_t** *thread***,**
    **pthread_attr_t *****attr_p***);**

## PARAMETERS

*thread* Specifies the thread for which you want the attribute object.

*attr_p* Receives the attribute object pointer returned by the call.

## DESCRIPTION

This fuction is used to get a thread attributes object for a specific thread. The attribute object obtained from this function call is used to set or get the individual attributes of a thread.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

This function returns 0 (zero) upon successful completion of the call.

**ERRORS**

If an error occurs, this function can return the following value:

[ESRCH] No thread could be found corresponding to the specified *thread* parameter.

**RELATED INFORMATION**

Functions: **pthread_attr_getstacksize(2)**, **pthread_attr_setstacksize(2)**, **pthread_attr_getstackaddr(2)**, **pthread_attr_setdetachstate(2)**, **pthread_attr_getdetachstate(2)**, **pthread_attr_setinheritsched(2)**, **pthread_attr_getinheritsched(2)**, **pthread_attr_setschedparam(2)**, **pthread_attr_getschedparam(2)**, **pthread_attr_getschedpolicy(2)**, **pthread_attr_setschedpolicy(2)**, **pthread_attr_getguardsize_np(2)**, **pthread_attr_setguardsize_np(2)**, **pthread_attr_init(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_getconcurrency** - Gets level of concurrency

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_getconcurrency(void);**

## DESCRIPTION

Concurrency values range from 0 to MAXINT inclusive. A concurrency level of 0 suggests to the scheduler that the minimum possible amount of concurrency is required. Concurrency levels greater than 0 suggest an increasingly higher level of concurrency.

The current implementation of concurrency level (Con Levl) and the minimum scheduled quantum is as follows:

| Con Levl | Minimum Scheduled Quantum |
|---|---|
| --------- | ---------------------- |
| 0 | Infinity |
| 1 | 1 second |
| 2 | 0.5 seconds |
| ... | ... |
| 10 | 0.1 seconds |
| ... | ... |
| 100 | 0.01 seconds |

Note that the quantum is calculated using the formula, *1 / concurrency_level*.

The default concurrency level for applications that use the POSIX User Thread Model library is 20; the default concurrency level for applications that use the Standard POSIX Threads library is 0.

The **pthread_setconcurrency( )** function does not support thread scheduling. The **pthread_setconcurrency( )** function checks for I/O completion when there is a context switch between threads and when the concurrency level is met.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

The **pthread_getconcurrency( )** function always returns the concurrency level set by a previous call to **pthread_setconcurrency( )**. If the **pthread_setconcurrency( )** function has never been called, **pthread_getconcurrency( )** returns zero.

## RELATED INFORMATION

Functions: **pthread_setconcurrency(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_get_expiration_np -** Calculates an absolute expiration time

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

```
int pthread_get_expiration_np(
        const struct timespec *delta,
        struct timespec *abstime);
```

## PARAMETERS

*delta* Specifies the number of seconds and nanoseconds to add to the current system time to determine an expiration time.

*abstime* Receives the calculated absolute expiration time. The resulting *abstime* value is in Universal Coordinated Time (UTC).

## DESCRIPTION

This function adds a specified interval to the current absolute system time and returns a new absolute time, which is used as the expiration time in a call to the **pthread_cond_timedwait( )** function.

The **timespec** structure contains the following two fields:

**tv_sec** Is an integral number of seconds.

**tv_nsec** Is an integral number of nanoseconds.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running

H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

On successful completion, this function returns a 0 (zero). If an error condition occurs, this function returns -1 and sets **errno** to indicate the type of error.

**ERRORS**

The following error conditions can occur:

[EINVAL] The value specified by the *delta* parameter is invalid.

**RELATED INFORMATION**

Functions: **pthread_cond_timedwait(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 Version 2 specification and to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

## NAME

**pthread_getschedparam -** Obtains the current scheduling policy and scheduling parameters of a thread

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

```
int pthread_getschedparam(
        pthread_t thread,
        int *policy,
        struct sched_param *param);
```

## PARAMETERS

*thread* Specifies the thread whose scheduling policy and parameters are obtained.

*policy* Receives the value of the scheduling policy for the thread specified by the *thread* parameter. See the **pthread_setschedparam(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for valid parameter values and their meanings.

*param* Receives the value of the scheduling parameters for the thread specified by the *thread* parameter. See the **pthread_setschedparam(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for valid values.

## DESCRIPTION

This function obtains both the current scheduling policy and associated scheduling parameters of the thread specified by the *thread* parameter.

The priority value returned in the structure specified by the *param* parameter is the value specified in the *attr* parameter passed to the **pthread_create( )** function or by the most recent call to the **pthread_setschedparam( )** function that affected this thread.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## NOTES

This function differs from the **pthread_attr_getschedpolicy( )** and **pthread_attr_getschedparam( )** functions, which get the scheduling policy and parameters that are used to establish the priority and scheduling policy of a new thread when it is created. **pthread_getschedparam( )** obtains the scheduling policy and parameters of an existing thread.

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values ares:

| | |
|---|---|
| 0 | Successful completion. |
| [ESRCH] | The *thread* parameter does not refer to an existing thread. |

## RELATED INFORMATION

Functions: **pthread_attr_getschedparam(2)**, **pthread_attr_getschedpolicy(2)**, **pthread_create(2)**, **pthread_self(2)**, **pthread_setschedparam(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_getspecific -** Obtains the thread-specific data associated with a key

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**void *pthread_getspecific(**
    **pthread_key_t** *key***);**

## PARAMETERS

*key* Specifies the context key that identifies the thread-specific data to be obtained.

## DESCRIPTION

This function obtains the thread-specific data bound to the key specified by the *key* parameter for the calling thread. Obtain this key by calling the **pthread_key_create( )** function.

This function can be called from a thread-specific data-destructor routine.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

This function returns the thread-specific data associated with *key*. If no thread-specific data is associated with *key* or if *key* is not defined, then this function returns a **NULL** value.

## RELATED INFORMATION

Functions: **pthread_key_create(2)**, **pthread_setspecific(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_get_threadstateinfo_np -** Gets the thread state information

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

```
#include <pthread.h> | #include <spthread.h>
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

int pthread_get_threadstateinfo_np(
        pthread_t *tid,
        char *state);
```

## PARAMETERS

*tid* Specifies the thread ID for which the state information is to be fetched.

*state* Specifies a pointer to the buffer where the thread state information is to be stored. The buffer must be at least 15 bytes in length.

## DESCRIPTION

The **pthread_get_threadstateinfo_np( )** obtains the state information of the thread identified by the *tid* parameter. On successful completion, the buffer contains a string that indicates the current state of the thread. Possible string values are:

**RUNNING** The thread is running.

**READY** The thread is ready to run.

**BLOCKED** The thread is blocked.

**TERMINATED**
The thread is terminated.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If an error condition occurs, the **pthread_get_threadstateinfo_np( )** function returns an integer value to identify the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The *tid* parameter does not specify an existing thread. |

## RELATED INFORMATION

Functions: **pthread_create(2)**, **pthread_detach(2)**, **pthread_join(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification and to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

## NAME

**pthread_join -** Causes the calling thread to wait for the termination of a thread

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_join(**
    **pthread_t** *thread***,**
    **void *****value_ptr***);**

## PARAMETERS

*thread* Specifies the thread whose termination is awaited by the calling thread.

*value_ptr* Receives the return value of the terminating thread.

## DESCRIPTION

This function suspends execution of the calling thread until the thread specified by the *thread* parameter terminates.

On return from a successful **pthread_join( )** call with a non-**NULL** value for *value_ptr*, the value passed to the **pthread_exit( )** function is returned in the location specified by *value_ptr* and the terminating thread is detached.

A call to **pthread_join( )** returns after the thread *thread* terminates.

The **pthread_join( )** function is a deferred cancelation point: the thread *thread* is not detached if the thread blocked in **pthread_join( )** is canceled. If the thread that is being joined is canceled, the **pthread_join( )** function returns **PTHREAD_CANCELED** in the *value_ptr* parameter.

If the calling thread specifies itself as the *thread* value, [EDEADLK] is returned. A deadlock does not occur.

The **pthread_join( )** or **pthread_detach( )** function should eventually be called for every thread that is created with the **detachstate** attribute of its thread attributes object set to **PTHREAD_CREATE_JOINABLE**, so that storage associated with the thread can be reclaimed.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## NOTES

If more than one thread attempts to join with the same thread, the results are unpredictable.

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EDEADLK] | A deadlock was detected, or the *thread* parameter specifies the calling thread. |
| [EINVAL] | The *thread* parameter does not specify a joinable thread. |
| [ESRCH] | The *thread* parameter does not specify an existing thread identifier. |

## RELATED INFORMATION

Functions: **pthread_cancel(2)**, **pthread_create(2)**, **pthread_detach(2)**, **pthread_exit(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_key_create -** Generates a unique thread-specific data key

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_key_create(**
    **pthread_key_t ***key**,**
    **void (***destructor***)(void *));**

## PARAMETERS

*key* Specifies the location where the new thread-specific data key is to be stored.

*destructor* Specifies a routine called to destroy a thread-specific data value associated with the created key when a thread terminates. The argument to the destructor for the user-specified routine is the non-**NULL** value associated with a key.

## DESCRIPTION

This function generates a unique thread-specific data key that is visible to all threads in the process. The key provided by this function is an opaque object used to locate thread-specific data. Although the same key value can be used by different threads, the values bound to the key by the **pthread_setspecific( )** function are maintained on a per-thread basis and persist for the life of the calling thread.

The system imposes a maximum number of thread-specific data keys, equal to the symbolic constant **PTHREAD_KEYS_MAX**.

Thread-specific data allows client software to associate static information with the current thread. For example, where a routine declares a variable **static** in a single-threaded program, a multithreaded version of the program might create a thread-specific data key to store the same variable.

This function generates and returns a new key value. The key reserves a cell within each thread. Each call to this function creates a new cell that is unique within an application invocation. Keys must be generated from initialization code that is guaranteed to be called only once within each process. (See the **pthread_once(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for more information.)

When a thread terminates, its thread-specific data is automatically destroyed; however, the key remains unless it is destroyed by a call to the **pthread_key_delete( )** function. An optional destructor routine can be associated with each key. At thread exit, if a key is associated with a non-**NULL** *destructor* parameter, and if the thread has a non-**NULL** value associated with that key, the destructor routine is called with the current associated value as its only argument. The order in which thread-specific data destructors are called at thread termination is undefined.

Before each destructor routine is called, the thread's value for the corresponding key is set to **NULL**. When each destructor routine is called, the destructor must release all storage associated with the key; otherwise, the destructor will be called again. After the destructor routines have been called for all non-**NULL** values with associated destructor routines, if some non-**NULL** values with associated destructor routines still exist, then this sequence of actions is repeated. This sequences is repeated up to **PTHREAD_DESTRUCTOR_ITERATIONS** times. If, after all allowed repetitions of this sequence, non-**NULL** values for any key with a destructor routine exist, the system terminates the thread. At this point, any key values that represent allocated heap are lost.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |

[EAGAIN] The system lacked the necessary resources to create another thread-specific data key, or the limit on the total number of keys for a process (**PTHREAD_KEYS_MAX**) has been exceeded.

[ENOMEM] Insufficient memory exists to create the key.

## RELATED INFORMATION

Functions: **pthread_getspecific(2)**, **pthread_key_delete(2)**, **pthread_once(2)**, **pthread_setspecific(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_key_delete -** Deletes a thread-specific data key

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_key_delete(**
    **pthread_key_t** *key***);**

## PARAMETERS

*key* Specifies the thread-specific data key to be deleted.

## DESCRIPTION

This function deletes the thread-specific data key specified by the *key* parameter, which must have been previously returned by the **pthread_key_create( )** function.

The thread-specific data values associated with the specified key need not be **NULL** at the time this function is called. The application must free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads. This cleanup can be done either before or after this function is called.

No destructor routines are invoked by this function. Any destructor routines that might have been associated with the specified key are not called upon thread exit. **pthread_key_delete( )** can be called from within destructor routines.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**NOTES**

Do not attempt to use the deleted key after calling this function; unpredictable behavior results.

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified for the *key* parameter is invalid. |

**RELATED INFORMATION**

Functions: **pthread_exit(2)**, **pthread_getspecific(2)**, **pthread_key_create(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_kill -** Sends a signal to a thread

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */
**#include <signal.h>**

**int pthread_kill(**
    **pthread_t** *thread***,**
    **int** *sig***);**

## PARAMETERS

*thread* Specifies the thread to receive the signal.

*sig* Specifies the signal to send. The valid values for this parameter are described in the **signal(4)** reference page available either online or in the *Open System Services System Calls Reference Manual*.

## DESCRIPTION

This function provides a mechanism for asynchronously directing a signal to a thread within the calling process. Per-thread signals have the following characteristics:

- Each signal is handled in the context of the specified thread. However, the signal action (terminating or stopping) affects the entire process.
- Signal action should be manipulated using the **sigaction( )** function instead of the **signal( )** function.
- Job control signals are not supported. The stop/continue-a-process actions implied by these signals is not supported.
- Signals send using **pthread_kill( )** are queued in first-in/first-out (FIFO) order for the target thread; more than one instance of the same signal can be pending for a thread. However, applications should not rely on this ordering.
- The realtime signals extension option is not supported.
- Whether a signal generates a saveabend file can be controled using a compiler or linker option.
- If a signal is delivered to a thread that is waiting on a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it had not been interrupted. The thread is not unblocked with a 0 (zero) return code.

Specifying a *sig* value of 0 (zero) causes this function to validate the *thread* parameter but not to send any signal.

If this function does not execute successfully, no signal is sent.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**NOTES**

The name of this function is misleading, because many signals do not terminate a thread.

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |

[EINVAL] The value of the *sig* parameter is invalid or specifies an unsupported signal.

[ESRCH] The *thread* parameter does not specify an existing thread.

**RELATED INFORMATION**

Functions: **pthread_sigmask(2), sigaction(2)**, **spt_alarm(2)**, **spt_sigaction(2)**, **spt_signal(2)**, **spt_sigpending(2)**, **spt_sigsuspend(2)**, **spt_sigwait(2)**. Files: **signal(4)**.

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 Version 2 specification.

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_kill_np -** Cancels a thread if a specified signal is received

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */
**#include <signal.h>**

**int pthread_kill_np(**
    **pthread_t** *thread***,**
    **int** *sig***);**

## PARAMETERS

*thread* Specifies the thread to receive the signal.

*sig* Specifies the signal to send. The valid values for this parameter are described on the **signal(4)** reference page available either online or in the *Open System Services System Calls Reference Manual*.

## DESCRIPTION

The **pthread_kill_np( )** function provides a mechanism for asynchronously directing a signal to a thread within the calling process. This function provides a similar functionality to the **pthread_kill( )** function, but it does not yield the processor and it is safe to call from a signal handler.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

### RETURN VALUES

If an error condition occurs, the **pthread_kill_np( )** function returns an integer value to identify the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value of the *sig* parameter is invalid or specifies an unsupported signal. |
| [ESRCH] | The *thread* parameter does not specify an existing thread. |

### RELATED INFORMATION

Functions: **pthread_kill(2)**, **pthread_sigmask(2)**, **sigaction(2)**, **spt_alarm(2)**, **spt_sigaction(2)**, **spt_signal(2)**, **spt_sigpending(2)**, **spt_sigsuspend(2)**, **spt_sigwait(2)**.

### STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification and to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

## NAME

**pthread_lock_global_np -** Locks the global mutex for threads

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_lock_global_np(void);**

## DESCRIPTION

This function locks the threads global mutex. If the threads global mutex is currently held by another thread when this function is called, the calling thread waits for the threads global mutex to become available and then locks it.

The thread that locks the threads global mutex becomes its current owner and remains its owner until the same thread unlocks it. This function returns with the threads global mutex in the locked state and the calling thread as the threads global mutex's current owner.

Use the threads global mutex when calling a library package that is not designed to run in a multithreaded environment. Unless documentation specifically states that a function is thread-safe, assume that the function is not compatible; in other words, assume it is nonreentrant.

The threads global mutex is one lock. Any code that calls any function that is not known to be reentrant should use the same lock to prevent problems resulting from dependencies among threads that call library functions, those functions' calling other functions, and so on.

The threads global mutex is a recursive mutex. A thread that locks the threads global mutex can relock it without deadlocking. The locking thread must call the **pthread_unlock_global_np()** function as many times as it called this function, to allow another thread to lock the threads global mutex.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

Possible return values are as follows:

0 Successful completion.

**RELATED INFORMATION**

Functions: **pthread_unlock_global_np(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 Version 2 specification and to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

## NAME

**pthread_mutexattr_destroy -** Destroys a mutex attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <pthread.h> | #include <spthread.h>
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

int pthread_mutexattr_destroy(
        pthread_mutexattr_t *attr);
```

## PARAMETERS

*attr* Specifies the mutex attributes object to be destroyed.

## DESCRIPTION

This function destroys a mutex attributes object by unintializing it. Call this function when your program no longer needs the specified mutex attributes object.

After this function is called, the system might reclaim the storage used by the destroyed mutex attributes object. Destroying a mutex attributes object does not affect any mutexes that were previously created using that mutex attributes object.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**NOTES**

The functions to set and get the process shared attribute are not supported by this implementation.

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *attr* parameter is invalid. |

**RELATED INFORMATION**

Functions: **pthread_mutexattr_init(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_mutexattr_getkind_np -** Obtains the mutex type attribute of a mutex attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <spthread.h>

extern int pthread_mutexattr_getkind_np(
        pthread_mutexattr_t attr );
```

## PARAMETERS

*attr* specifies the mutex attributes object whose mutex type is to be obtained.

## DESCRIPTION

The **pthread_mutexattr_getkind_np( )** function obtains the mutex type attribute of the mutex attributes object specified by the *attr* parameter. See the **pthread_mutexattr_setkind_np(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the mutex type attribute.

## RETURN VALUES

One of the following values can be returned:

0 Successful completion.

[EINVAL] The value specified by the *attr* parameter is invalid.

## RELATED INFORMATION

Functions: **pthread_mutex_init(2)**, **pthread_mutexattr_init(2)**, **pthread_mutexattr_setkind_np(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification and to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

## NAME

**pthread_mutexattr_gettype -** Gets the mutex **type** attribute of a mutex attribute object

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/yputdll**

## SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_gettype(
        const pthread_mutexattr_t *attr,
        int *type);
```

## PARAMETERS

*attr* Specifies the address of a pthread mutex attribute object whose mutex **type** attribute is to be obtained.

*type* Specifies the mutex type based on the **type** value retrieved from the *attr* parameter value.

## DESCRIPTION

The **pthread_mutexattr_gettype( )** function gets the mutex **type** attribute of the mutex attribute object specified by the *attr* parameter.

The following are the valid values for the *type* parameter and their characteristics:

**PTHREAD_MUTEX_NORMAL**
A normal mutex does not detect deadlock. If a thread attempts to relock a normal mutex without first unlocking it, the thread deadlocks. If a thread attempts to unlock a normal mutex locked by a different thread, the result is undefined behavior. If a thread attempts to unlock an unlocked normal mutex, the result is undefined behavior.

**PTHREAD_MUTEX_ERRORCHECK**
An error checking mutex provides error checking. If a thread attempts to relock an error checking mutex without first unlocking it, the function returns an error. If a thread attempts to unlock an error checking mutex locked by a different thread, the function returns an error. If a thread attempts to unlock an unlocked error checking mutex, the function returns an error.

**PTHREAD_MUTEX_RECURSIVE**
A recursive mutex can be locked more than once by the same thread without returning an error. If a thread attempts to relock a recursive mutex without first unlocking it, the thread succeeds in locking the mutex. The relocking deadlock that can occur with normal mutexes cannot occur with this type of mutex. Multiple locks of a recursive mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. If a thread attempts to unlock a recursive mutex that another thread has locked, the function returns an error. If a thread attempts to unlock an unlocked recursive mutex, the function returns an error.

**PTHREAD_MUTEX_DEFAULT**
A default mutex can be locked only once. If a thread attempts to relock a default mutex, the result is undefined behavior. If a thread attempts to unlock a default mutex locked by a different thread, the result is undefined behavior. If a thread attempts to unlock an unlocked default mutex, the result is undefined behavior. An implementation can map this mutex to one of the other mutex types.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**NOTES**

This function is not supported with the Standard POSIX Threads (SPT) library. SPT-based applications should use the **pthread_mutexattr_getkind_np( )** function instead.

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment using the POSIX User Thread Model library, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **pthread_mutexattr_gettype( )** function returns 0 (zero) and stores the retrieved value of the *type* parameter; otherwise, the function returns an error number to identify the error.

**ERRORS**

If the **pthread_mutexattr_gettype( )** function call fails, **errno** may return the following value:

[EINVAL] The value specified by the *attr* parameter is invalid.

This function does not return the [EINTR] error code.

**RELATED INFORMATION**

Functions: **pthread_mutex_init(2)**, **pthread_mutexattr_getkind_np(2)**, **pthread_mutexattr_init(2)**, **pthread_mutexattr_settype(2)**.

### STANDARDS CONFORMANCE

This function conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**pthread_mutexattr_init -** Initializes a mutex attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_mutexattr_init(**
**pthread_mutexattr_t** ***attr*);**

## PARAMETERS

*attr* Specifies the mutex attributes object to be initialized.

For the Standard POSIX Threads library, if the value specified is **pthread_mutexattr_default**, then the default attribute is:

**MUTEX_FAST_NP**
specifies the mutex type (normal). A normal mutex is locked exactly once by a thread. If a thread tries to lock the mutex again without first unlocking it, the thread waits for itself to release the lock and thereby deadlocks.

## DESCRIPTION

This function initializes the mutex attributes object specified by the *attr* parameter with a set of default attribute values. A mutex attributes object is used to specify the attributes of mutexes when they are created. The mutex attributes object created by this function is used only in calls to the **pthread_mutex_init( )** function.

When a mutex attributes object is used to create a mutex, the values of the individual attributes determine the characteristics of the new mutex. Thus, mutex attributes objects act as additional arguments to creation of mutexes. Changing individual attributes in a mutex attributes object does not affect any mutexes that were previously created using that mutex attributes object.

You can use the same mutex attributes object in successive calls to **pthread_mutex_init( )** from any thread. If multiple threads can change attributes in a shared mutex attributes object, your program must use a mutex to protect the integrity of that mutex attributes object.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

### NOTES

The functions to set and get the process shared attribute are not supported by this implementation.

### RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [ENOMEM] | Not enough memory exists to initialize the mutex attributes object. |

### RELATED INFORMATION

Functions: **pthread_mutex_init(2)**, **pthread_mutexattr_destroy(2)**.

### STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_mutexattr_setkind_np -** Sets the mutex type attribute of a mutex attributes object

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <spthread.h>

extern int pthread_mutexattr_setkind_np(
        pthread_mutexattr_t *attr,
        int kind );
```

## PARAMETERS

*attr* specifies the mutex attributes object whose mutex type attribute is to be modified.

*kind* specifies the new value for the mutex type attribute. Valid values are:

**MUTEX_FAST_NP**
This is the default value. Creates a default mutex.

**MUTEX_NONRECURSIVE_NP**
Creates a normal mutex.

**MUTEX_RECURSIVE_NP**
Creates a recursive mutex.

## DESCRIPTION

The **pthread_mutexattr_setkind_np( )** function sets the mutex type attribute of the mutex attributes object specified by the *attr* parameter.

A fast (default) mutex is locked and unlocked in the fastest manner possible. A fast mutex can be locked (obtained) only once. All subsequent calls to the **pthread_mutex_lock( )** function by the owning thread return [EDEADLK]. Subsequent calls by another thread block.

A normal (nonrecursive) mutex is locked only once by a thread, like a fast (default) mutex. If the thread tries to lock the mutex again without first unlocking it, the thread receives an error. Also, if someone other than the owner tries to unlock a nonrecursive mutex, an error is returned.

A recursive mutex can be locked more than once by the same thread without returning an error. That is, a single thread can make consecutive calls to **pthread_mutex_lock( )** without blocking. The thread must then call the **pthread_mutex_unlock( )** function the same number of times as it called **pthread_mutex_lock( )** before another thread can lock the mutex.

Never use a recursive mutex with condition variables, because the implicit unlock performed for a call to the **pthread_cond_wait( )** or **pthread_cond_timedwait( )** function might not actually release the mutex. In that case, no other thread can satisfy the condition of the predicate.

## RETURN VALUES

One of the following values can be returned:

0 Successful completion.

[EINVAL] The value specified by the *attr* parameter is invalid.

[EPERM] The caller does not have the appropriate privileges to perform this operation.

[ERANGE] One or more of the parameters have an invalid value.

## RELATED INFORMATION

Functions: **pthread_mutex_init(2)**, **pthread_mutexattr_getkind_np(2)**, **pthread_mutexattr_init(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification and to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

## NAME

**pthread_mutexattr_settype -** Sets the mutex **type** attribute of a mutex attribute object

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**int pthread_mutexattr_settype(**
  **pthread_mutexattr_t ****attr***,**
  **int** *type***);**

## PARAMETERS

*attr* Specifies the address of a pthread mutex attribute object whose mutex **type** attribute is to be updated.

*type* Specifies the mutex type that is used to update the **type** attribute for the *attr* parameter value.

## DESCRIPTION

The **pthread_mutexattr_settype( )** function sets the mutex **type** attribute of the mutex attribute object specified by the *attr* parameter.

The following are the valid *type* parameter values and their characteristics:

**PTHREAD_MUTEX_NORMAL**
A normal mutex does not detect deadlock. If a thread attempts to relock a normal mutex without first unlocking it, the thread deadlocks. If a thread attempts to unlock a normal mutex locked by a different thread, the result is undefined behavior. If a thread attempts to unlock an unlocked normal mutex, the result is undefined behavior.

**PTHREAD_MUTEX_ERRORCHECK**
An error checking mutex returns error checking. If a thread attempts to relock an error checking mutex without first unlocking it, the function returns an error. If a thread attempts to unlock an error checking mutex locked by a different thread, the function returns an error. If a thread attempts to unlock an unlocked error checking mutex, the function returns an error.

**PTHREAD_MUTEX_RECURSIVE**
A recursive mutex can be locked more than once by the same thread without returning an error. If a thread attempts to relock a recursive mutex without first unlocking it, the thread succeeds in locking the mutex. The relocking deadlock that can occur with normal mutexes cannot occur with this type of mutex. Multiple locks of a recursive mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. If a thread attempts to unlock a recursive mutex that another thread has locked, the function returns an error. If a thread attempts to unlock an unlocked recursive mutex, the function returns an error.

**PTHREAD_MUTEX_DEFAULT**
A default mutex can be locked only once. If a thread attempts to relock a default mutex, the result is undefined behavior. If a thread attempts to unlock a default mutex locked by a different thread, the result is undefined behavior. If a thread attempts to unlock an unlocked default mutex, the result is undefined behavior. An implementation can map this mutex to one of the other mutex types.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**NOTES**

This function is not supported with the Standard POSIX Threads (SPT) library. SPT-based applications should use the **pthread_mutexattr_setkind_np( )** function instead.

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment using the POSIX User Thread Model library, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **pthread_mutexattr_settype( )** function returns 0 (zero); otherwise, the function returns an error number to identify the error.

**ERRORS**

If the **pthread_mutexattr_settype( )** function call fails, **errno** may return the following value:

[EINVAL] The value specified by the *attr* or *type* parameter is invalid.

This function does not return the [EINTR] error code.

**RELATED INFORMATION**

Functions: **pthread_mutex_init(2)**, **pthread_mutexattr_gettype(2)**, **pthread_mutexattr_init(2)**, **pthread_mutexattr_setkind_np(2)**.

**STANDARDS CONFORMANCE**

This function conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**pthread_mutex_destroy -** Destroys a mutex

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_mutex_destroy(**
    **pthread_mutex_t ****mutex***);**

## PARAMETERS

*mutex* Specifies the mutex to be destroyed.

## DESCRIPTION

This function destroys the specified mutex by uninitializing it. Call this function when your program no longer needs the specified mutex object.

After this function is called, the system might reclaim the storage used by the destroyed mutex.

Destroying an initialized mutex that is unlocked is safe. Destroying a locked mutex is not allowed.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**NOTES**

The results of this function are unpredictable if the mutex object specified by the *mutex* parameter does not exist or is not initialized.

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EBUSY] | An attempt was made to destroy the mutex indicated by the *mutex* parameter while it is locked or referenced. |
| [EINVAL] | The value specified for the *mutex* parameter is invalid. |

**RELATED INFORMATION**

Functions: **pthread_mutex_init(2)**, **pthread_mutex_lock(2)**, **pthread_mutex_trylock(2)**, **pthread_mutex_unlock(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_mutex_init** - Initializes a mutex

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_mutex_init(**
**pthread_mutex_t ****mutex***,**
**const pthread_mutexattr_t ****attr***);**

## PARAMETERS

*mutex* Specifies the mutex to be initialized.

*attr* Specifies the mutex attributes object that defines the characteristics of the mutex to be initialized.

## DESCRIPTION

This function initializes a mutex with the attributes of the mutex attributes object specified by the *attr* parameter. A mutex is a synchronization object that allows multiple threads to serialize their access to shared data.

A mutex is a resource of the process, not part of any particular thread. A mutex is neither destroyed nor unlocked automatically when any thread exits. If a mutex is allocated on a stack, static initializers cannot be used on the mutex.

The mutex is initialized and set to the unlocked state. If *attr* is **NULL**, the default mutex attributes are used.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running

H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**NOTES**

Use the **PTHREAD_MUTEX_INITIALIZER** macro to statically initialize a mutex without calling this function. Use this macro as follows:

**pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;**

Only normal mutexes can be statically initialized.

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error, the mutex is not initialized, and the contents of mutex are undefined. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EAGAIN] | The system lacks the necessary resources to initialize the mutex. |
| [EBUSY] | The system detected an attempt to reinitialize a mutex (an attempt to initialize a previously initialized but not yet destroyed mutex). |
| [EINVAL] | The value specified by the *attr* parameter is invalid. |
| [ENOMEM] | Insufficient memory exists to initialize the mutex. |
| [EPERM] | The caller does not have the required privileges to perform the operation. |

**RELATED INFORMATION**

Functions: **pthread_mutex_destroy(2)**, **pthread_mutex_lock(2)**, **pthread_mutex_trylock(2)**, **pthread_mutex_unlock(2)**, **pthread_mutexattr_destroy(2)**, **pthread_mutexattr_init(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_mutex_lock -** Locks an unlocked mutex

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_mutex_lock(**
**pthread_mutex_t ****mutex***);**

## PARAMETERS

*mutex* Specifies the mutex to be locked.

## DESCRIPTION

This function locks a mutex. The result depends upon the type of mutex:

- If *mutex* is a fast or nonrecursive mutex, an error is returned if the current owner of the mutex calls this function in an attempt to lock the mutex a second time.
- If *mutex* is a recursive mutex, the current owner of the mutex can relock the same mutex without blocking. The lock count is incremented for each recursive lock within the thread.

The thread that locks a mutex becomes its current owner and remains its owner until the same thread unlocks it. This function returns with the mutex in the locked state and the current thread as the mutex's current owner.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

### RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EDEADLK] | The current thread already owns the mutex. |
| [EINVAL] | The value specified by the *mutex* parameter is not a valid mutex. |
| [ENOMEM] | There is insufficient memory to lock a statically initialized mutex. |

### RELATED INFORMATION

Functions: **pthread_mutex_destroy(2)**, **pthread_mutex_init(2)**, **pthread_mutex_trylock(2)**, **pthread_mutex_unlock(2)**.

### STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

The return of [ENOMEM] is an HP extension to the POSIX standard.

## NAME

**pthread_mutex_trylock -** Attempts to lock a specified mutex but does not wait if the mutex is already locked

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_mutex_trylock(**
    **pthread_mutex_t *** *mutex***);**

## PARAMETERS

*mutex* Specifies the mutex to be locked.

## DESCRIPTION

This function attempts to lock the mutex specified by the *mutex* parameter. When a thread calls this function, an attempt is made to immediately lock the mutex. If the mutex is successfully locked, this function returns 0 (zero) and the calling thread becomes the mutex's current owner. If the specified mutex is already locked, the calling thread does not wait for the mutex to become available and the function returns.

This function does the following:

- If *mutex* is a fast or nonrecursive mutex: if the mutex is already locked by any thread (including the calling thread) when this function is called, this function returns [EBUSY] and the calling thread does not wait to acquire the lock.
- If *mutex* is a recursive mutex: if the mutex is either unlocked or owned by the calling thread, this function returns 0 (zero) and the mutex lock count is incremented. (To unlock a recursive mutex, each lock must be matched by a call to the **pthread_mutex_unlock( )** function.)

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EBUSY] | The mutex is already locked; therefore, it was not acquired. |
| [EINVAL] | The value specified by the *mutex* parameter is not a valid mutex. |
| [ENOMEM] | There is insufficient memory to lock a statically initialized mutex. |

## RELATED INFORMATION

Functions: **pthread_mutex_destroy(2)**, **pthread_mutex_init(2)**, **pthread_mutex_lock(2)**, **pthread_mutex_unlock(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

The return of [ENOMEM] is an HP extension to the POSIX standard.

## NAME

**pthread_mutex_unlock -** Unlocks a mutex

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_mutex_unlock(**
**pthread_mutex_t** ***mutex*);**

## PARAMETERS

*mutex* Specifies the mutex to be unlocked.

## DESCRIPTION

This function unlocks the mutex specified by the *mutex* parameter. This function does the following:

- If *mutex* is a fast or nonrecursive mutex: if the mutex is owned by the calling thread, it is unlocked with no current owner. If the mutex is not locked or is already locked by another thread, [EPERM] is returned.
- If *mutex* is a recursive mutex: if the mutex is owned by the calling thread, the lock count is decremented. The mutex remains locked and owned until the lock count reaches 0 (zero). When the lock count reaches 0 (zero), the mutex becomes unlocked with no current owner.

If one or more threads are waiting to lock the specified mutex and the mutex becomes unlocked, this function causes one thread to unblock and try to acquire the mutex. The scheduling policy is used to determine which thread to unblock. A blocked thread is chosen in priority order, using a first-in/first-out (FIFO) algorithm within priorities. Note that the mutex might not be acquired by the unblocked thread if another running thread attempts to lock the mutex first.

If a signal is delivered to a thread waiting for a mutex, then upon return from the signal handler, the thread resumes waiting for the mutex as if it had not been interrupted.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified for the *mutex* parameter is invalid. |
| [EPERM] | The calling thread does not own the mutex. |

**RELATED INFORMATION**

Functions: **pthread_mutex_destroy(2)**, **pthread_mutex_init(2)**, **pthread_mutex_lock(2)**, **pthread_mutex_trylock(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_once -** Calls a routine to be executed once by a single thread

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**pthread_once_t once_control = PTHREAD_ONCE_INIT;**

**int pthread_once(**
  **pthread_once_t ****once_control***,**
  **void (****routine***)(void));**

## PARAMETERS

*once_control* Specifies a block that controls the one-time execution code. Each one-time execution routine must have its own unique **pthread_once_t** block.

*routine* Specifies the address of the routine to be executed once. This routine is called only once, regardless of the number of times it and its associated *once_control* block are passed to **pthread_once( )**.

## DESCRIPTION

The first call to this function by any thread in a process with a given *once_control* block calls the routine specified by *routine* with no arguments. Subsequent calls to **pthread_once( )** with the same *once_control* block do not call the routine. On return from **pthread_once( )**, the routine is guaranteed to have finished.

For example, a mutex or a per-thread context key must be created exactly once. Calling **pthread_once( )** ensures that the initialization is serialized across multiple threads. Other threads that reach the same point in the code are delayed until the first thread is finished.

To initialize the *once_control* block, use the **PTHREAD_ONCE_INIT** macro, as shown in the **SYNOPSIS**.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## NOTES

If you specify a routine that directly or indirectly results in a recursive call to **pthread_once( )** and that specifies the same routine argument, the recursive call can result in a deadlock.

## EXAMPLES

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified for the *once_control* parameter is not valid. |

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

**NAME**

**pthread_self -** Obtains the thread identifier of the calling thread

**LIBRARY**

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

**SYNOPSIS**

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**pthread_t pthread_self(void);**

**DESCRIPTION**

This function returns the thread identifier of the calling thread.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the

following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

This function returns the thread identifier of the calling thread.

**RELATED INFORMATION**

Functions: **pthread_cancel(2)**, **pthread_create(2)**, **pthread_detach(2)**, **pthread_exit(2)**, **pthread_join(2)**, **pthread_kill(2)**, **pthread_sigmask(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_setcancelstate -** Sets the calling thread's cancelability state

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_setcancelstate(**
**int** *state***,**
**int ****oldstate***);**

## PARAMETERS

*state* Specifies the new cancelability state for the calling thread. Valid values are:

**PTHREAD_CANCEL_ENABLE**
**PTHREAD_CANCEL_DISABLE**

*oldstate* Receives the previous cancelability state for the calling thread.

## DESCRIPTION

This function sets the calling thread's cancelability state to the value of the *state* parameter and returns its previous cancelability state in the *oldstate* parameter.

When the cancelability state is set to **PTHREAD_CANCEL_DISABLE**, a cancelation request cannot be delivered to the thread, even if a cancelable routine is called or an asynchronous cancelability type is enabled.

When a thread is created, the default cancelability state is **PTHREAD_CANCEL_ENABLE**.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

### Possible Problems When Disabling Cancelability

The most important use of thread cancelation is to ensure that possibly indefinite wait operations are terminated. For example, a thread that waits on some network connection, which can possibly take days to respond (or might never respond), should be made cancelable.

When a thread's cancelability is disabled, no routine in that thread is cancelable. As a result, the user is unable to cancel the operation performed by that thread. When disabling cancelability, be sure that no long waits can occur and that no cancelation requests must be deferred around that particular region of code for other reasons.

## RETURN VALUES

On successful completion, this function returns the calling thread's previous cancelability state in the *oldstate* parameter.

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The specified cancelability state is not **PTHREAD_CANCEL_ENABLE** or **PTHREAD_CANCEL_DISABLE**. |

## RELATED INFORMATION

Functions: **pthread_cancel(2)**, **pthread_setcanceltype(2)**, **pthread_testcancel(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_setcanceltype -** Sets the calling thread’s cancelability type

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_setcanceltype(**
**int** *type***,**
**int** ***oldtype***);**

## PARAMETERS

*type* Specifies the cancelability type to set for the calling thread. Valid values are:

**PTHREAD_CANCEL_DEFERRED**

*oldtype* Receives the previous cancelability type for the calling thread.

## DESCRIPTION

This function sets the calling thread’s cancelability type to the value of the *type* parameter and returns its previous cancelability type in the *oldtype* parameter.

When the cancelability state is **PTHREAD_CANCEL_DISABLE** (see the **pthread_setcancelstate(2)** reference page either online or in the *Open System Services System Calls Reference Manual*), a cancelation request cannot be delivered to the thread, even if a cancelable routine is called or the asynchronous cancelability type is enabled.

When the cancelability state is **PTHREAD_CANCEL_ENABLE**, cancelability depends on the thread’s cancelability type. If the thread’s cancelability type is **PTHREAD_CANCEL_DEFERRED**, the thread can receive a cancelation request only at specific cancelation points (including condition waits, thread joins, and calls to the **pthread_testcancel( )** function.)

When a thread is created, the default cancelability type is **PTHREAD_CANCEL_DEFERRED**. The cancelability type of **PTHREAD_CANCEL_ASYNCHRONOUS** is not supported in this implementation.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## NOTES

If the asynchronous cancelability type is set, do not call any function unless it is explicitly documented as safe to be called with the asynchronous cancelability type. The only safe functions are **pthread_setcanceltype( )** and **pthread_setcancelstate( )**.

The asynchronous cancelability type should be used only when you have a compute-bound section of code that carries no state and makes no function calls.

## RETURN VALUES

On successful completion, this function returns the calling thread's previous cancelability type in the *oldtype* parameter.

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The specified type is not **PTHREAD_CANCEL_DEFERRED** or **PTHREAD_CANCEL_ASYNCHRONOUS**. |
| [ENOTSUP] | The specified type is **PTHREAD_CANCEL_ASYNCHRONOUS**. |

## RELATED INFORMATION

Functions: **pthread_cancel(2)**, **pthread_setcancelstate(2)**, **pthread_testcancel(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

The use of [ENOTSUP] is an HP extension to the POSIX standard.

## NAME

**pthread_setconcurrency -** Sets level of concurrency

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_setconcurrency(int new_level);**

## DESCRIPTION

The **pthread_setconcurrency( )** does not support thread scheduling. This function checks for I/O completion when there is a context switch between threads and when the concurrency level is met.

Concurrency values range from 0 to **MAXINT** inclusive. A concurrency level of 0 suggests to the scheduler that the minimum possible amount of concurrency is required. Concurrency levels greater than 0 suggest an increasingly higher level of concurrency.

The current implementation of concurrency level (Con Levl) and the minimum scheduled quantum is as follows:

| Con Levl | Minimum Scheduled Quantum |
|---|---|
| --------- | ---------------------------- |
| 0 | Infinity |
| 1 | 1 second |
| 2 | 0.5 seconds |
| ... | ... |
| 10 | 0.1 seconds |
| ... | ... |
| 100 | 0.01 seconds |

Note that the quantum is calculated using the formula, *1 / concurrency_level*.

The default concurrency level for applications that use the POSIX User Thread Model library is 20; the default concurrency level for applications that use the Standard POSIX Threads library is 0.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If successful, the **pthread_setconcurrency( )** function returns 0 (zero). Otherwise, an error number is returned to indicate the error.

## ERRORS

The **pthread_setconcurrency( )** function will fail if:

[EINVAL] The value specified by new_level is negative.

[EAGAIN] The value specific by new_level would cause a system resource to be exceeded.

## RELATED INFORMATION

Functions: **pthread_getconcurrency(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

**NAME**

**pthread_setschedparam** - Sets the scheduling policy and scheduling parameters of a thread

**LIBRARY**

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

**SYNOPSIS**

```
#include <pthread.h> | #include <spthread.h>
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

int pthread_setschedparam(
        pthread_t thread,
        int policy,
        const struct sched_param *param);
```

**PARAMETERS**

*thread* Specifies the thread whose scheduling policy and parameters are to be set.

*policy* Specifies the new scheduling policy value for the thread specified by the *thread* parameter. Valid values are:

**SCHED_FIFO**

*param* Specifies one or more new values for the scheduling parameters associated with the scheduling policy specified by the *policy* parameter. Valid values for the **sched_priority** field of a **sched_param** structure depend on the chosen scheduling policy. Use the **sched_get_priority_min( )** and **sched_get_priority_max( )** functions to determine the low and high limits of each scheduling policy. The default priority is 24.

**DESCRIPTION**

This function sets both the current scheduling policy and scheduling parameters of the thread specified by the *thread* parameter to the policy and parameters provided by the *policy* and *param* parameters, respectively.

The scheduling policies of all threads have one scheduling attribute named **sched_priority**. For the scheduling policy you choose, you must specify an appropriate value in the **sched_priority** field of the **sched_param** structure.

Changing the scheduling policy, priority, or both, of a thread can cause it to start executing or to be preempted by another thread. A thread sets its own scheduling policy and priority by using the handle returned by the **pthread_self( )** function.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## NOTES

This function differs from the **pthread_attr_setschedpolicy( )** and **pthread_attr_setschedparam( )** functions, which set the scheduling policy and scheduling parameters used to establish the priority and scheduling policy of a new thread when it is created. However, **pthread_setschedparam( )** sets the scheduling policy and scheduling parameters of an existing thread.

## RETURN VALUES

If an error condition occurs, no scheduling policy or parameters are changed for the thread *thread*, and this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *policy* or *param* parameter is invalid. |
| [ENOTSUP] | An attempt was made to set the scheduling policy or a scheduling parameter to an unsupported value. |

[ESRCH] The *thread* parameter does not refer to an existing thread.

**RELATED INFORMATION**

Functions: **pthread_attr_setschedparam(2)**, **pthread_attr_setschedpolicy(2)**, **pthread_create(2)**, **pthread_self(2)**, **sched_yield(2)**.

**STANDARDS CONFORMANCE**

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

with the following exception:

- [EPERM] is not returned.

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_setspecific -** Sets the thread-specific data associated with a key

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_setspecific(**
    **pthread_key_t** *key***,**
    **const void *** *value***);**

## PARAMETERS

*key* Specifies the thread-specific key that identifies the thread-specific data to be set to the new value. This key value is returned by the **pthread_key_create( )** function.

*value* Specifies the new thread-specific data to associate with the key specified by *key*.

## DESCRIPTION

This function sets the thread-specific data associated with the key specified by the *key* parameter for the calling thread.

Different threads can bind different data to the same key. This data typically consists of pointers to blocks of dynamically allocated memory that are reserved for use by the calling thread.

Although the data type of the *value* parameter (**void ***) implies that it represents an address, the type is being used as a "universal scalar type." The system simply stores the value of *value* for later retrieval.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running

H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The specified key is invalid. |
| [ENOMEM] | Insufficient memory exists to associate the new data with the key. |

## RELATED INFORMATION

Functions: **pthread_getspecific(2)**, **pthread_key_create(2)**, **pthread_key_delete(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_sigmask -** Examines or changes the calling thread's signal mask

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <pthread.h> | #include <spthread.h>
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */
[#include <signal.h>]

int pthread_sigmask(
        int how,
        const sigset_t *set,
        sigset_t *oset);
```

## PARAMETERS

*how* Indicates how the set of masked signals is to be changed. Valid values are:

**SIG_BLOCK** The resulting set is the union of the previous set and the signal set indicated by the *set* parameter.

**SIG_SETMASK**
The resulting set is the signal set indicated by the *set* parameter.

**SIG_UNBLOCK**
The resulting set is the intersection of the previous signal set and the complement of the signal set indicated by the *set* parameter.

*set* Specifies a signal set by pointing to a set of signals used to change the blocked set. If this value is **NULL**, the *how* parameter is ignored and the signal mask is unchanged.

*oset* Receives the value of the current signal mask (unless this value is **NULL**).

## DESCRIPTION

This function examines or changes the calling thread's signal mask. Typically, you use the **SIG_BLOCK** value for the *how* parameter to block signals during a critical section of code, and then use the **SIG_SETMASK** value for the *how* parameter to restore the signal mask to the value returned by the previous call to **pthread_sigmask( )**.

If any unblocked signals are pending after a call to this function, at least one of those signals is delivered before this function returns.

This function does not allow the **SIGKILL** or **SIGSTOP** signals to be blocked. If a program attempts to block one of these signals, **pthread_sigmask( )** gives no indication of the error.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified for the *how* parameter is invalid. |

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 Version 2 specification.

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_signal_to_cancel_np -** Cancels a thread if a specified signal is received

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_signal_to_cancel_np(**
    **sigset_t ***_sigset_**,**
    **pthread_t ***_thread_**);**

## PARAMETERS

*sigset* Specifies a signal mask containing a list of signals that, when received by the thread, cancel the specified thread.

*thread* Specifies the thread to be canceled if a specified signal is received by the thread.

## DESCRIPTION

The **pthread_signal_to_cancel_np( )** function requests that the thread specified by the *thread* parameter be canceled if one of the signals in the signal mask specified by the *sigset* parameter is received by the process. The set of signals that can be specified is the same as the set for the **sigwait( )** function.

The *sigset* parameter is not validated. If it is invalid, this function returns successfully but neither the specified thread nor any previously specified thread is canceled if a signal occurs.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

### NOTES

The address of the specified thread is saved in a per-process global variable. Therefore, any subsequent call to this function by your application or any library function replaces the thread specified in the previous call, and that thread is not canceled if one of the signals specified for it is delivered to the process. Be careful when you call this function; if another thread calls it after you do, the expected result of this function might not occur.

### RETURN VALUES

One of the following values can be returned:

| | |
|---|---|
| 0 | Successful completion. |
| [EINVAL] | The value specified by the *thread* parameter is invalid. |

### RELATED INFORMATION

Functions: **pthread_cancel(2)**, **sigwait(2)**.

### STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification and to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

## NAME

**pthread_testcancel -** Requests delivery of a pending cancelation request to the calling thread

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**void pthread_testcancel(void);**

## DESCRIPTION

This function requests delivery of a pending cancelation request to the calling thread. Thus, calling this function creates a cancelation point within the calling thread.

The cancelation request is delivered only if a request is pending for the calling thread and the calling thread's cancelability state is enabled. (A thread disables delivery of cancelation requests to itself by calling the **pthread_setcancelstate( )** function.)

When called within very long loops, this function ensures that a pending cancelation request is noticed by the calling thread within a reasonable amount of time.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RELATED INFORMATION

Functions: **pthread_setcancelstate(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**pthread_unlock_global_np -** Unlocks the threads global mutex

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int pthread_unlock_global_np(void);**

## DESCRIPTION

This function unlocks the threads global mutex. Because the threads global mutex is recursive, the unlock occurs when the number of calls to this function match the number of calls to the **pthread_lock_global_np( )** function. For example, if you called **pthread_lock_global_np( )** three times, then the third time you call **pthread_unlock_global_np( )**, it unlocks the threads global mutex.

If no threads are waiting for the threads global mutex, it becomes unlocked with no current owner. If one or more threads are waiting to lock the threads global mutex, this function causes one thread to unblock and try to acquire the threads global mutex. The scheduling policy determines which thread is unblocked. For the policies **SCHED_FIFO** and **SCHED_RR**, a blocked thread is chosen in priority order, using a first-in/first-out (FIFO) algorithm within priorities.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

### RETURN VALUES

If an error condition occurs, this function returns an integer value indicating the type of error. Possible return values are:

| | |
|---|---|
| 0 | Successful completion. |
| [EPERM] | The threads global mutex is unlocked or owned by another thread. |

### RELATED INFORMATION

Functions: **pthread_lock_global_np(2)**.

### STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification and to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

**NAME**

**put_awaitio -** Awaits a tagged I/O file

**LIBRARY**

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

**SYNOPSIS**

**#include <pthread.h>**

**put_error_t put_awaitio(**
    **const short** *filenum***,**
    **const long** *tag***,**
    **const long** *timelimit***,**
    **long ****count_transferred***,**
    **int ****error***,**
    **void ****userdata***);**

**PARAMETERS**

*filenum* Specifies Guardian file number being waited on.

*tag* Specifies tag being waited on.

*timelimit* Specifies how many hundredths of a second to wait for a completed I/O:

| | |
|---|---|
| -1 | means wait forever |
| 0 | means immediate return |

*count_transferred*
Specifies transfer count of completed I/O; set by callback when **PUT_SUCCESS** is returned.

*error* Specifies Guardian error number for I/O; set by callback when **PUT_SUCCESS** is returned or as described in **ERRORS**.

*userdata* Specifies address of user data area; the referenced data may be modified by a callback.

**DESCRIPTION**

Awaits a tagged I/O on file number to complete, timeout, or be interrupted (see the **put_interrupt(2)** reference page under **RETURN VALUES**). The function never cancels I/O. I/O completes only if **PUT_SUCCESS** is returned. Multiple threads should not await the same tagged I/O on any given file number.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**NOTES**

While using this API on a filenum obtained from **PUT_FILE_OPEN_**, the tag parameter must be the same as the filenum parameter passed.

**RETURN VALUES**

**PUT_SUCCESS**
File number was waited on.

**PUT_ERROR** An error occurred. See **ERRORS**.

**PUT_TIMEDOUT**
Time limit has expired. See **ERRORS**.

**PUT_INTERRUPTED**
Wait was interrupted. See **ERRORS**.

**ERRORS**

16 *filenum* is not registered.

29 *filenum* < 0 (zero).

40 *timelimit* has expired.

[EINTR] Wait was interrupted via **put_interrupt( )**, **put_interruptTag( )**, or a signal was received via **pthread_kill( )** and is not blocked, ignored, or handled.

## NAME

**PUT_CANCEL -** Cancels the oldest incomplete operation on a Guardian file opened for nowait I/O

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_CANCEL(
        short filenum);
```

## PARAMETERS

*filenum* Specifies the Guardian file number of a Guardian file open instance whose oldest incomplete operation you want to cancel.

## DESCRIPTION

The **PUT_CANCEL( )** function is a thread-aware version of the Guardian CANCEL procedure.

The **PUT_CANCEL( )** function is used to cancel the oldest incomplete operation on a Guardian file opened for nowait I/O. The canceled operation might or might not have had effects. For disk files, the file position might or might not be changed.

For programming information about the Guardian CANCEL procedure, see the *Guardian Programmer's Guide*.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

### Considerations

Queue files — If a **PUT_READUPDATELOCKX( )** function operation is canceled using the **PUT_CANCEL( )** function, the **PUT_READUPDATELOCKX( )** call might already have deleted a record from the queue file, which could result in the loss of a record from the queue file. For audited queue files only, your application can recover from a timeout error by calling the **PUT_ABORTTRANSACTION( )** function, when detecting Guardian file-system error 40, to ensure that any dequeued records are reinserted into the file.

For nonaudited queue files, there is no recovery of a lost record. Thus, your application should never call the Guardian AWAITIOX procedure with a time limit greater than 0 (zero) if a **PUT_READUPDATELOCKX( )** call is pending. The **PUT_ABORTTRANSACTION( )** recovery procedure does not work on nonaudited queue files.

Messages — The server process (that is, a process that was opened and to which the I/O request was sent) receives a system message -38 (queued message cancellation) that identifies the canceled I/O request, if it has requested receipt of such messages. If the server has already replied to the I/O request, message -38 is not delivered. For details about system message -38, see the *Guardian Procedure Errors and Messages Manual*.

## RETURN VALUES

The **PUT_CANCEL( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**PUT_CONTROL -** Performs device-dependent input/output operations

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_CONTROL(
        short filenum,
        short operation,
        short param,
        long tag);
```

## PARAMETERS

*filenum* Specifies the Guardian file number of a Guardian file open instance, identifying the file on which the underlying CONTROL procedure performs an input or output operation.

*operation* Specifies a value from 1 through 27 that defines a type of operation to be performed. For tables that list *operation* numbers and the possible *param* values for each, see the description of the CONTROL procedure in the *Guardian Procedure Calls Reference Manual*.

*param* (Optional) Specifies a value that defines the operation to be performed. For tables that list *operation* numbers and the possible *param* values for each, see the description of the CONTROL procedure in the *Guardian Procedure Calls Reference Manual*.

*tag* (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **PUT_CONTROL( )** function is a thread-aware version of the Guardian CONTROL procedure. CONTROL is used to perform device-dependent input or output operations.

If the **PUT_CONTROL( )** function is used on a file that is opened for nowait I/O, the function must be completed with a call to the AWAITIO procedure.

The following considerations apply to use on disk files:

Writing EOF to an unstructured file
Writing EOF to an unstructured disk file sets the EOF pointer to the relative byte address indicated by the setting of the next-record pointer and writes the new EOF setting in the file label on disk. (File pointer action for CONTROL operation 2, write EOF.)

File is locked If a CONTROL operation is attempted for a file locked through a *filenum* other than that specified in the call to **PUT_CONTROL( )**, the call is rejected with a "file is locked" error 73. If any record is locked in a file, a call to **PUT_CONTROL( )** to write EOF (operation 2) to that same file will be rejected with a "file is locked" error 73.

The following considerations apply to use on magnetic tapes:

When device is not ready
: If a magnetic tape rewind is performed concurrently with application program execution (that is, a rewind operation other than 6), any attempt to perform a read, write, or control operation to the rewinding tape unit while rewind is taking place results in an error indication. A subsequent call to the FILE_GETINFO_ or FILEINFO procedure shows that an error 100 occurred.

Wait for rewind to complete
: If a magnetic tape rewind operation of 6 (wait for completion) is performed as a nowait operation, the application waits at the call to the AWAITIO procedure for the rewind to complete.

The following considerations apply to use for interprocess communication:

Nonstandard *operation* and *param* values
: You can specify any value for the *operation* and *param* parameters. An application-defined protocol should be established for interpreting nonstandard parameter values.

Process not accepting system messages
: If the object of the control operation is not accepting process CONTROL messages, the call to **PUT_CONTROL()** completes but a subsequent call to the FILE_GETINFO_ or FILEINFO procedure shows that an error 7 occurred.

Process control
: You can obtain the process identifier of the caller to **PUT_CONTROL()** in a subsequent call to the FILE_GETRECEIVEINFO_ (or LASTRECEIVE or RECEIVEINFO) procedure.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.

- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**RETURN VALUES**

The **PUT_CONTROL( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

When device handlers do not allow the operation, Guardian file-system error 2 is returned. For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

**ERRORS**

None. This function does not set the **errno** variable.

**RELATED INFORMATION**

Functions: **PUT_CANCEL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**put_fd_read_ready -** Waits on read-ready file descriptor

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <pthread.h>

int put_fd_read_ready(
        const int fd,
        struct timeval *timeout);
```

## PARAMETERS

*fd* Specifies an OSS file descriptor.

*timeout* On input, the maximum interval to wait for *fd* ready; if NULL, then no timeout will occur. On output, the interval remaining.

## DESCRIPTION

The **put_fd_read_ready** function waits on a file descriptor to be read-ready or have an exception pending.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

0 (zero) No error.

[EINTR] A signal was received via **pthread_kill( )** and is not blocked, ignored, or handled.

[EINVAL] Invalid function argument.

[EBADF] File descriptor not open for reading or closed while being waited on.

[ENOTSUP] Operation not supported on file descriptor.

[ETIMEDOUT]
The timeout has occurred.

## NAME

**put_fd_write_ready** - Waits on write-ready file descriptor

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**int put_fd_write_ready(**
    **const int** *fd***,**
    **struct timeval ****timeout***);**

## PARAMETERS

*fd* Specifies an OSS file descriptor.

*timeout* On input, specifies the maximum interval to wait for *fd* ready.

If NULL, specifies that no *timeout* will occur.

On output, specifies the interval remaining.

## DESCRIPTION

The **put_fd_write_ready** function waits on a file descriptor to be write-ready or have an exception pending.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

0 (zero) No error.

[EINTR] A signal was received via **pthread_kill( )** and is not blocked, ignored, or handled.

[EINVAL] Invalid function argument.

[EBADF] File descriptor was not open for writing or was closed while being waited on.

[ENOTSUP] Operation was not supported on file descriptor.

[ETIMEDOUT]
*timeout* has occurred.

## NAME

**PUT_FILE_CLOSE_ -** Closes an open Guardian file

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <pthread.h>**

**short PUT_FILE_CLOSE_(**
**short** *filenum***,**
**short** *tape_disposition***);**

## PARAMETERS

*filenum* Specifies the file number of a Guardian file open instance that identifies the file to be closed.

*tape_disposition*
(Optional) Indicates the tape control action to take:

0 Rewind and unload; do not wait for completion.

1 Rewind and unload, do not wait for completion.

2 Rewind and leave online; do not wait for completion.

3 Rewind and leave online; wait for completion.

4 Do not rewind; leave online.

Other input values result in no error if the file is a tape device; the control action might be unpredictable. If this parameter is omitted, 0 (zero) is used.

## DESCRIPTION

The **PUT_FILE_CLOSE_ ( )** function is a thread-aware version of the Guardian FILE_CLOSE_ procedure.

The FILE_CLOSE_ procedure closes a Guardian file open instance. Closing a file open instance terminates access to the file through that open instance. You can use **PUT_FILE_CLOSE_( )** to close files that were opened by **PUT_FILE_OPEN_( )**.

For programming information about the FILE_CLOSE_ procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

### Considerations

Returning space allocation after closing a file
: Closing a disk file causes the space that is used by the resident file control block to be returned to the system main-memory pool if the disk file is not open concurrently. A temporary disk file is purged if the file was not open concurrently. Any space that is allocated to that file is made available for other files. With any file closure, the space allocated to the access control block (ACB) is returned to the system.

Closing a nowait file open
: If a **PUT_FILE_CLOSE_( )** call is executed for a nowait file that has pending operations, any incomplete operations are canceled. There is no indication as to whether the operation completed or not.

Labeled tape processing
: If your system has labeled tape processing enabled, all tape actions (as specified by *tape_disposition*) wait for completion.

Process close message
: A process can receive a process close system message when it is closed by another process. It can obtain the process handle of the closer by a subsequent call to the Guardian FILE_GETRECEIVEINFO_ procedure. For detailed information about system messages, see the *Guardian Procedure Errors and Messages Manual*.

  This message is also received if the close is made by the backup process of a process pair. Therefore, a process can expect two of these messages when being closed by a process pair.

## RETURN VALUES

The **PUT_FILE_CLOSE_ ( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEX(2)**.

## NAME

**PUT_FILE_OPEN_ -** Establishes a communication path between an application process and a file

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <pthread.h>**

**short PUT_FILE_OPEN_(**
  {**const char ***filename* | **const char****pathname*}
  **short** *length***,**
  **short ***filenum***,**
  **short** *access***,**
  **short** *exclusion***,**
  **short** *nowait_depth***,**
  **short** *sync_or_receive_depth***,**
  **short** *options***,**
  **short** *seq_block_buffer_id***,**
  **short** *seq_block_buffer_len***,**
  **short ***primary_processhandle***,**
  **int** *elections***);**

## PARAMETERS

*filename | pathname*

The *filename* parameter specifies the Guardian filename of a Guardian file to be opened. The value of *filename* must be a valid fully or partially qualified file name or DEFINE name. If the name is partially qualified, it is resolved using the contents of the =_DEFAULTS DEFINE.

The *pathname* parameter specifies the OSS filename or pathname of an OSS file to be opened. The value of the *pathname* parameter is terminated by a null character. *options* bit 10 must be set to 1 to open an OSS file.

*filenum*

Returns a Guardian file number that is used to identify the Guardian file open instance in subsequent Guardian file-system calls. If the file cannot be opened, a value of -1 is returned.

The *filenum* parameter is used as an input parameter only when you are attempting a backup open. In that case, you must supply the *primary_processhandle* parameter or else the input value of *filenum* is ignored. For a backup open, the value specified for *filenum* must be the *filenum* value that was returned when the file was opened by the primary process. If a backup open is successful, the input value of *filenum* is returned unless *options* bit 3 is set, in which case a new file number is assigned for the backup open. If the backup open is unsuccessful, -1 is returned.

*access*

Specifies the desired access mode for the file to be opened. Valid values are:

| | |
|---|---|
| 0 | Read-write |

| | |
|---|---|
| 1 | Read only |
| 2 | Write only |
| 3 | Extend (supported only for tape) |

The default is 0 (zero).

*exclusion* Specifies the desired mode of compatibility with other openers of the file. Valid values are:

| | |
|---|---|
| 0 | Shared |
| 1 | Exclusive |
| 2 | Process exclusive |
| 3 | Protected |

The default is 0 (zero).

*nowait_depth* Specifies the number of nowait I/O operations that can be in progress for the file concurrently with other processing. If this parameter is omitted or 0 (zero), only waited I/O operations are permitted against the file. The maximum value is 1 for disk files and $RECEIVE. The maximum value is 15 for other objects, except for the Transaction Monitoring Facility (TMF) transaction pseudofile (TFILE), which has a maximum of 1000. For details about the TFILE, see the *TMF Application Programmer's Guide*.

*sync_or_receive_depth*
The purpose of this parameter depends on the type of device being opened:

disk file Specifies the number of nonretryable (that is, write) requests whose completion the Guardian file system must remember. You must specify a value of 1 or greater to recover from a path failure occurring during a write operation. This value also implies the number of write operations that the primary process in a process pair can perform to this file without intervening checkpoints to its backup process. For disk files, this parameter is called sync depth. The maximum value is 15.

If omitted, or if 0 (zero) is specified, internal checkpointing does not occur. Disk path failures are not automatically retried by the file system.

$RECEIVE file
Specifies the maximum number of incoming messages read by the **PUT_READUPDATEX( )** function that the application process is allowed to queue before corresponding reply operations must be performed. If omitted or 0 (zero), **PUT_READUPDATEX( )** and reply operations to $RECEIVE are not permitted. For $RECEIVE, this parameter is called receive depth, and the maximum number of queued incoming messages is 4047 in H06.17 and J06.06 and earlier RVUs. For H06.18 and J06.07 and later RVUs, the maximum receive depth value is increased from 4047 to 16300.

process pair — Specifies whether an I/O operation is automatically redirected to the backup process if the primary process or its processor module fails. For processes, this parameter is called sync depth. The process determines the maximum value. The value must be at least 1 for an I/O operation to a remote process pair to recover from a network failure. If this parameter is greater than or equal to 1, the server is expected to save or be able to regenerate that number of replies. If this parameter is 0 (zero), and if an I/O operation cannot be performed to the primary process of a process pair, an error indication is returned to the originator of the message. On a subsequent I/O operation, the file system redirects the request to the backup process.

For other device types, the meaning of this parameter depends on whether the sync-ID mechanism is supported by the device being opened. If the device does not support the sync-ID mechanism, 0 (zero) is used regardless of what you specify (this is the most common case). If the device supports the sync-ID mechanism, specifying a nonzero value causes the results of that number of operations to be saved; in case of path failures, the operations are retried automatically. The actual value being used can be obtained by a call to the FILE_GETINFOLIST_ procedure.

*options* — Specifies optional characteristics as a bit mask. The bits, when set to 1, indicate:

0 — Unstructured access. For disk files, access is to occur as if the file were unstructured, that is, without regard to record structures and partitioning. (For unstructured files, setting this bit to 1 causes secondary partitions to be inaccessible.) This bit must be 0 (zero) for other devices.

1 — Nowait open processing. Specifies that the processing of the open proceeds in a nowait manner. Unless **PUT_FILE_OPEN_( )** returns an error, a nowait open must be completed by a call to the Guardian AWAITIOX procedure. This option cannot be specified for the TMF transaction pseudofile (TFILE). This option does not determine the nowait mode of I/O operations. The *nowait_depth* parameter, which controls the nowait mode of I/O operations, must have a nonzero value when you use this option.

2 — No open time update. For disk files, the "time of last open" file attribute is not updated by this open. This bit must be 0 (zero) for other devices.

3 — Any file number for backup open. When performing a backup open, specifies that the system can use any file number for the backup open. A value of 0 (zero) specifies that the backup open is to have the same file number as the primary open. Guardian file-system error 12 is returned if that file number is already in use.

4 through 9 — Reserved; specify 0 (zero).

| | |
|---|---|
| 10 | Open an OSS file by its OSS pathname. Specifies that the file to be opened is identified by the *pathname* parameter. |
| 11 | Reserved; specify 0 (zero). |
| 12 | No transactions. For $RECEIVE, messages are not to include transaction identifiers. This bit must be 0 (zero) if bit 15 is 1. |
| 13 | Internationalization locale support. For $RECEIVE, data messages include internationalization locale information. This bit must be 0 (zero) if bit 15 is 1. For information about internationalization, see the *Software Internationalization Guide*. |
| 14 | Old-format system messages. For $RECEIVE, system messages should be delivered in C-series format. If this bit is 0 (zero), D-series format messages are delivered. For other device types, this bit must be 0 (zero). See **Interprocess Communication Considerations** in the **DESCRIPTION** subsection of this reference page. |
| 15 | No file-management system messages. For $RECEIVE, specifies that the caller does not wish to receive process open, process close, CONTROL, SETMODE, SETPARAM, RESETSYNC, and CONTROLBUF messages. If this bit is 0 (zero), messages are delivered as normal; some messages are received only with **PUT_SETMODE(80)**. For other device types, this bit must be 0 (zero). |

When *options* is omitted, 0 (zero) is used for all bits.

*seq_block_buffer_id*

If present and not 0 (zero), identifies the buffer to be used for shared sequential block buffering; all opens made through **PUT_FILE_OPEN_( )** and using this ID share the same buffer. You can supply any integer value for this parameter.

If *seq_block_buffer_id* is omitted or 0 (zero), and sequential block buffering is requested, the buffer is not shared. In this case, the buffer resides in the process's process file segment (PFS) with the size given by *seq_block_buffer_len*.

*seq_block_buffer_len*

Specifies whether sequential block buffering is being requested. If this parameter is supplied with a value greater than 0 (zero), it indicates a request for sequential block buffering and specifies the length in bytes of the sequential block buffer. If this parameter is omitted or 0 (zero), sequential block buffering is not requested. Sequential block buffering is only for disk files.

If this value is less than the data-block length that was given to this file or to any associated alternate-key file, the larger value is used. Supplying a nonzero value for this parameter causes a buffer to be allocated unless an existing buffer is to be shared (see the *seq_block_buffer_id* parameter). If an existing buffer is to be shared, but it is smaller than *seq_block_buffer_len*, sequential block buffering is not provided and a warning value of 5 is returned.

*primary_processhandle*
Indicates that the caller is requesting a backup open and specifies the process handle of the primary process that already has the file open when its backup attempts to open the file. If this parameter is supplied and not null (a null process handle has -1 in each word), *filenum* must contain the *filenum* value that was returned to the primary. If a null process handle is supplied, or the parameter is omitted, a normal open is being requested. Use this option only when the backup process is the caller. It is more common for the primary process to perform this operation by a call to the FILE_OPEN_CHKPT_ procedure.

*elections*
Specifies the following options as a bit mask:

| | |
|---|---|
| 0 through 30 | Reserved; specify 0 (zero). |
| 31 | Use 64-bit primary keys. For disk files only, bit 31 specifies that 64-bit primary-key values are used instead of 32-bit values for unstructured, relative, or entry-sequenced files. Bit 31 is ignored for key-sequenced files and nondisk devices. |

You can use the *elections* parameter with both Format 1 and Format 2 Guardian files. If this parameter is omitted, 0 (zero) is used for all bits.

**DESCRIPTION**

The **PUT_FILE_OPEN_( )** function is a thread-aware version of the Guardian FILE_OPEN_ procedure.

The **PUT_FILE_OPEN_( )** function establishes a communication path between an application process and a file. When **PUT_FILE_OPEN_( )** successfully completes, it returns a Guardian file number to the caller. The file number identifies this access path to the file in subsequent Guardian file-system calls.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.

- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**General Considerations**

File numbers
: File numbers are unique within a process. The lowest file number is 0 (zero) and is reserved for $RECEIVE; the remaining file numbers start at 1. The lowest available file number is always assigned, except in the case of backup opens. When a file is closed, its file number becomes available for a subsequent file open to use.

Maximum number of open files
: The maximum number of files in the system that can be open at any given time depends on the space available for control blocks: access control blocks (ACBs), file control blocks (FCBs), and open control blocks (OCBs). The amount of space available for control blocks is limited primarily by the physical memory size of the system. The maximum amount of space for ACBs is determined by the size of the process file segment (PFS). See the description of the *pfs-size* parameter for the PROCESS_CREATE_ procedure in the *Guardian Procedure Calls Reference Manual*.

Multiple opens by the same process
: If a given file is opened more than once by the same process, a unique file number is returned for each open. These file numbers provide logically separate accesses to the same file; each file number has its own ACB, its own file position, and its own last error value. If a nowait I/O operation haS begun and a second nowait operation is started (using a second file number for the same file), the I/O requests:

- Are independent
- Might arrive in either order at the destination
- Might complete in either order

Multiple opens on a given file can create a deadlock. Locks are granted on an open file (that is, file number) basis. Therefore, if a process opens the same file multiple times, a lock of one file number excludes access to the file through other file numbers. The process is suspended forever if the default locking mode is in effect and a deadlock occurs.

Limit on number of concurrent opens
: There is a limit on the total number of concurrent opens permitted on a file. This determination includes opens by all processes. The specific limit for a file depends on the file's device type:

| | |
|---|---|
| Disk files | Cannot exceed 32,767 opens per disk. |
| Process | Defined by the process (see the discussion of controlling openers in the *Guardian Programmer's Guide*). |
| $0 | Unlimited opens. |
| $0.#ZSPI | 128 concurrent opens permitted. |
| $OSP | Ten times the number of subdevices (up to a maximum of 830 opens). |

$RECEIVE One open per process is permitted.

Other Varies by subsystem.

Specifying a *nowait_depth* value greater than 0 (zero) causes all I/O operations to be performed in a nowait manner. Nowait I/O operations must be completed by a call to the AWAITIOX procedure.

Nowait I/O operations on different file numbers (even if for the same file) are independent, might arrive in any order at the destination, and might be completed by AWAITIOX in any order.

Nowait opens If you open a file in a nowait manner (*options* bit 1 = 1) and if **PUT_FILE_OPEN_( )** returns no error (returns a value of 0 [zero]), the open operation must be completed by a call to AWAITIOX.

If there is an error, no system message is sent to the object being opened and you do not need to call AWAITIOX to complete the operation. If there is no error, the *filenum* parameter returned by **PUT_FILE_OPEN_( )** is valid; however, you cannot initiate any I/O operation on the file until you complete the open by calling AWAITIOX.

If you specify the *tag* parameter in the call to AWAITIOX, a -30D is returned; the values returned in the *buffer* and *count* parameters to AWAITIOX are undefined. If an error returns from AWAITIOX, it is your responsibility to close the file.

For the TMF transaction pseudofile, or for a waited file (*nowait_depth* = 0 [zero]), a request for a nowait open is rejected.

The Guardian file system implementation of a nowait open might use waited calls in some cases. However, it is guaranteed that the open message is sent using nowait I/O to a process; the opener does not wait for the process being opened to service the open message.

Direct and buffered I/O transfers

A file opened by **PUT_FILE_OPEN_( )** uses direct I/O transfers by default; SETMODE 72 is used to force the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers. This behavior is unlike the obsolescent Guardian OPEN procedure call, which uses a PFS buffer for I/O transfers by default.

Sequential block buffering

Sequential block buffering is only supported for disk files. If you are using sequential block buffering, the file should usually be opened with protected or exclusive access. You can use shared access, but it is somewhat slower than the other access methods, and there might be concurrency problems. See the discussion of "Sequential Block Buffering" in the *Enscribe Programmer's Guide*.

Named processes

If you supply a process filename for a named process, it can represent any process with the same name. System messages are normally sent to the current primary process. The exception is when a named process supplies its own name to **PUT_FILE_OPEN_( )**. In that case, the name refers to the backup process and system messages are sent to the backup process.

A named process can be represented with or without a sequence number. **PUT_FILE_OPEN_( )** treats the two name forms differently:

- If you supply a process file name that includes a sequence number, the process must have a matching sequence number or the open fails with error 14. When retrying I/O on a process opened under such a name, the file system does not attempt to send messages to a possible backup process of the same name unless it has a matching sequence number. This behavior ensures that the named process is a true backup of the primary process.
- If you supply a process file name that does not include a sequence number, any process with a matching name can be opened and can be sent I/O retries. A newly created process that receives an I/O retry intended for another process of the same name will usually reject it with an error 60, but this behavior is under the control of the application.

Partitioned files

A separate FCB exists for each partition of a partitioned file. There is one ACB per accessor (as for single-volume files), but this ACB requires more main memory because it contains the information necessary to access all of the partitions, including the location and partial-key value for each partition.

Disk file open security check

When a disk file open is attempted, the system performs a security check. The accessor's (that is, the caller's) security level is checked against the file security level for the requested access mode, as follows:

for read access
: read security level is checked.

for write access
: write security level is checked.

for read-write access
: read and write security levels are checked.

A Guardian file has one of seven levels of security for each access mode. The owner of the file can set the security level for each access mode by using SETMODE function 1 or by using the File Utility Program (FUP) SECURE command. The following table shows the seven levels of security:

Table 5−1. Levels of Guardian File Security

| FUP Code | Program Value | Access Permitted |
|---|---|---|
| - | 7 | Local super ID only |
| U | 6 | Owner (local or remote), that is, any user with owner's ID |
| C | 5 | Member of owner's group (local or remote), that is, any member of owner's community |
| N | 4 | Any user (local or remote) |
| O | 2 | Owner only (local) |

| | | |
|---|---|---|
| G | 1 | Member of owner's group (local) |
| A | 0 | Any user (local) |

For a given access mode, the accessor's security level is checked against the file security level. File access is allowed or not allowed as shown in the following table. In this table, file security levels are indicated by FUP security codes. For a given accessor security level, a Y indicates that access is allowed to a file with the security level shown; an X indicates that access is not allowed.

Table 5–2. Allowed Guardian File Accesses

| **Accessor's Security Level** | **File Security Level** - | U | C | N | O | G | A |
|---|---|---|---|---|---|---|---|
| Super ID user, local access | Y | Y | Y | Y | Y | Y | Y |
| Super ID user, remote access | X | Y | Y | Y | X | X | X |
| Owner or owner's group manager, remote access | X | Y | Y | Y | X | X | X |
| Member of owner's group, remote access | X | X | Y | Y | X | X | X |
| Any other user, remote access | X | X | X | Y | X | X | X |
| Owner or owner's group manager, local access | X | Y | Y | Y | Y | Y | Y |
| Member of owner's group, local access | X | X | Y | Y | X | Y | Y |
| Any other user, local access | X | X | X | Y | X | X | Y |

If the caller to **PUT_FILE_OPEN_( )** fails the security check, the open fails with an error 48. You can obtain the security level of a file by a call to the Guardian FILE_GETINFOLIST[BYNAME]_ procedure, the FILEINFO procedure, or by the File Utility Program (FUP) INFO command.

If you are using the Safeguard product, this security information might not apply.

Tape file open access mode

The file system does not enforce read-only or write-only access for unlabeled tape, even though no error is returned if you specify one of these access modes when opening a tape file.

File open exclusion and access mode checking

When a file open is attempted, the requested access and exclusion modes are compared with those of any opens already granted for the file. If the attempted open is in conflict with other opens, the open fails with error 12. For a table that lists the possible current modes and requested modes, indicating whether an open succeeds or fails, see the description of the FILE_OPEN_ procedure in the *Guardian Procedure Calls Reference Manual*. For the Optical Storage Facility only, the "process exclusive" exclusion mode is also supported. Process exclusive is the same as exclusive for opens by other processes, but the same as shared for opens by the same process.

Protected exclusion mode

Protected exclusion mode has meaning only for disk files. For other files, specifying protected exclusion mode is equivalent to specifying shared exclusion mode.

**Disk File Considerations**

Maximum number of concurrent nowait operations

The maximum number of concurrent nowait operations permitted for an open of a disk file is 1. Attempting to open a disk file and specify a *nowait_depth* value greater than 1 causes **PUT_FILE_OPEN_( )** to fail with an error 28.

Unstructured files

File pointers after an open

After a disk file is opened, the current-record and next-record pointers begin at a relative byte address (RBA) of 0, and the first data transfer (unless positioning is performed) is from that location. After a successful open, the pointers are:

current-record pointer = 0D
next-record pointer = 0D

Sharing the same EOF pointer

If a given disk file is opened more than once by the same process, separate current-record and next-record pointers are provided for each open, but all opens share the same EOF pointer.

Structured files

Accessing structured files as unstructured files

The unstructured access option (*options* bit 0 = 1) permits a file to be accessed as an unstructured file. You must maintain the block format used by Enscribe if the file is be accessed again in its structured form. (HP reserves the right to change this block format at any time.) For information about Enscribe block formats, see the *Enscribe Programmer's Guide*.

For a file opened using the unstructured access option, a data transfer occurs to the position in the file specified by an RBA (instead of to the position indicated by a key address field or record number); the number of bytes transferred is that specified in the file-system procedure call (instead of the number of bytes indicated by the record format).

If a partitioned file, either structured or unstructured, is opened using the unstructured access option, only the first partition is opened. You must open the remaining partitions individually with separate calls to **PUT_FILE_OPEN_( )** (each call specifying unstructured access).

Accessing audited structured files as unstructured files is not allowed.

Current-state indicators after an open

After successful completion of an open, the current-state indicators have these values:

- The current position is that of the first record in the file by primary key.
- The positioning mode is approximate.
- The comparison length is 0.

If the Guardian READ procedure is called immediately after **PUT_FILE_OPEN_( )** for a structured file, READ reads the first record in the file; in a key-sequenced file, this is the first record by primary key. Subsequent reads, without intervening positioning, read the file sequentially (in a relative or entry-sequenced file) or by primary key (in a key-sequenced file) through the last record in the file. When a key-sequenced file is opened, the Guardian KEYPOSITION procedure is usually called before any subsequent Guardian I/O procedure call (such as READ, READUPDATE, or WRITE) to establish a position in the file.

Queue files

If the READUPDATELOCK operation is to be used, the value of the *sync_or_receive_depth* parameter must be 0 (zero). You can use a separate open for operations with *sync_or_receive_depth* greater than 0 (zero).

You cannot use sequential block buffering.

64-bit primary keys

In order to access non-key-sequenced files bigger than 4 gigabytes, you must set bit 31 of the **PUT_FILE_OPEN_( )** *elections* parameter. Use of this parameter allows the use of procedures using 32-bit primary keys (POSITION, KEYPOSITION, REPOSITION, GETSYNCINFO, and SETSYNCINFO) and the 32-bit key items of the FILE_GETINFOLIST_, FILEINFO, and FILERECINFO procedures.

**Considerations for Terminals**

The terminal used as the operator console should not be opened with exclusive access. If it is, console messages are not logged.

**Interprocess Communication Considerations**

Maximum concurrent nowait operations for an open of $RECEIVE

The maximum number of concurrent nowait operations permitted for an open of $RECEIVE is 1. Attempting to open $RECEIVE and to specify a value greater than 1 causes an error 28 to be returned.

When **PUT_FILE_OPEN_( )** completes

When process A attempts to open process B, **PUT_FILE_OPEN_( )** completes as follows:

- If process B has already opened $RECEIVE with file-management system messages disabled, the open call by process A completes immediately.
- If process B has opened $RECEIVE requesting file-management system messages enabled, the open call completes when process B reads the open message from process A by using READX, or if B uses READUPDATEX, the open call completes when process B replies to the open message (by using REPLYX).

  If process B has not yet opened $RECEIVE, the open by process A does not complete until process B opens $RECEIVE. Specifically, the open by process A completes as follows:

— When process B opens $RECEIVE with file-management system messages disabled, a waited open by process A completes immediately, but a nowait open by process A completes after the first read of $RECEIVE by process B.

— When process B opens $RECEIVE with file-management system messages enabled, the open call by process A completes when process B reads the open message from A by using READ[X], or if B uses READUPDATE[X], the open call completes when process B replies to the open message (by using REPLY[X]).

Message formats

When $RECEIVE is opened by **PUT_FILE_OPEN_( )**, system messages are delivered to the caller in D-series format unless messages in C-series format are requested by setting *options* bit 14 to 1. (No file-management system messages are delivered to the caller if *options* bit 15 is set to 1 when opening $RECEIVE.)

Messages from high-PIN processes

Opening $RECEIVE with **PUT_FILE_OPEN_( )** implies that the caller is capable of handling messages from processes with PINs greater than 255.

Opening $RECEIVE and being opened by a remote long-named process

A process that has a process name consisting of more than five characters will fail with an error 20 if it attempts to open a process on a remote node and the process it attempts to open:

- Used the **PUT_FILE_OPEN_( )** procedure to open $RECEIVE and requested that C-series format messages be delivered, or
- Used the Guardian OPEN procedure to open $RECEIVE.

Notification of this failure is not sent to the process reading $RECEIVE.

Opening an unconverted (C-series format) process from a high-PIN process

A high-PIN process cannot open an unconverted process unless the unconverted process has the HIGHREQUESTERS object-file attribute set. If a high-PIN process attempts to open a low-PIN process that does not have this attribute set, the high-PIN process receives file-system error 560.

### System Message

When a process is opened by either **PUT_FILE_OPEN_( )** or the Guardian OPEN procedure, it receives a process open message (unless it specified when opening $RECEIVE that it wants no messages). This message is in D-series format (message -103) or in C-series format (message -30), depending on what the receiving process specified when it opened $RECEIVE. This message is also received if the backup process of a process pair performs an open. Therefore, a process can expect two of these messages when being opened by a process pair.

You can obtain he process handle of the opener by a subsequent call to FILE_GETRECEIVEINFO_. For a description of the process open message see the *Guardian Procedure Errors and Messages Manual*.

### DEFINE Considerations

- The *filename* or *pathname* parameter can be a DEFINE name; **PUT_FILE_OPEN_( )** uses the file name given by the DEFINE as the name of the object to be opened. If you specify a CLASS TAPE DEFINE without the DEVICE attribute, the system selects the tape drive to be opened. A CLASS TAPE DEFINE has other effects when supplied to **PUT_FILE_OPEN_( )**. For more information about DEFINEs, see Appendix E of the *Guardian Procedure Calls Reference Manual*.
- If a supplied DEFINE name is a valid name but no such DEFINE exists, the procedure returns an error 198 (missing DEFINE).
- When performing a backup open of a file originally opened with a DEFINE, *filename* must contain the same DEFINE name. The DEFINE must exist and must have the same value as when the primary open was performed.

### Safeguard Considerations

For information on files protected by Safeguard, see the *Safeguard Reference Manual*.

### OSS Considerations

- To open an OSS file by its pathname, set *options* bit 10 to 1 and specify the *pathname* parameter.
- You can open OSS files only with shared exclusion mode.

## EXAMPLES

The open in the following example has the following defaults: waited I/O, exclusion mode (shared), access mode (read/write), sync depth (0).

**error = PUT_FILE_OPEN_ ( filename, filenum );**

## RETURN VALUES

The **PUT_FILE_OPEN_( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

This function can return any error number that the Guardian FILE_OPEN_ procedure call can return. It can also return the following error number:

| | |
|---|---|
| 12 | Callback has already been registered for this *filenum*. |

Some error numbers are warnings (that is, they indicate conditions that do not prevent the file from being opened); check the value returned for the *filenum* parameter to determine whether the file was opened successfully. Forexplanation of other error numbers returned, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**PUT_FILE_WRITEREAD_** - Writes data to a process previously opened from an array and waits for data to be transferred back from the process

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_FILE_WRITEREAD_ (
        short filenum,
        char *write_buffer,
        char *read_buffer,
        int write_count,
        int read_count,
        int *count_read,
        long tag);
```

## PARAMETERS

### Input

*filenum* Specifies the file number of a Guardian file open instance that identifies the file to be read.

*write_buffer* Specifies an array in the application process in which the information to be written to the file is stored before the call.

*write_count* Specifies the number of bytes to be written.

*read_count* Specifies the number of bytes to be read.

*tag* (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

### Output

*read_buffer* Specifies an array in the application process which contains the information read from the file.

*count_read* (Optional) For waited I/O only. This parameter returns a count of the number of bytes returned from the file into *read_buffer*.

## DESCRIPTION

The **PUT_FILE_WRITEREAD_( )** function is the thread-aware version of the Guardian FILE_WRITEREAD_ procedure and FILE_WRITEREAD64_ procedures.

The **PUT_FILE_WRITEREAD_( )** function writes data to a process, which was previously opened from an array in the application process, then waits for data to be transferred back from the process. The data buffers for the **PUT_FILE_WRITEREAD_( )** procedure can be either in the caller's stack segment or an extended data segment for the write portion.

If the file is opened for nowait I/O, you must not modify the *write_buffer* or *read_buffer* before the I/O completes with a call to the Guardian AWAITIOX procedure. This condition also applies to other processes that might be sharing the segment. The application must ensure that the buffers used in the call to the **PUT_FILE_WRITEREAD_( )** function are not reused before the

I/O completes with a call to AWAITIOX.

For programming information about the FILE_WRITEREAD_ and FILE_WRITEREAD64_ procedures, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

You can use this function with 32-bit applications or 64-bit applications on systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs.

To use this function in a 32-bit application, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

To use this function in a 64-bit application, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**Considerations**

Buffer use

**PUT_FILE_WRITEREAD_( )** is intended for use with 32-bit extended addresses and 64-bit extended addresses. The data buffers for **PUT_FILE_WRITEREAD_( )** can be either in the caller's stack segment or any extended data segment.

Interprocess communication

The **PUT_FILE_WRITEREAD_( )** function is used to originate a message to another process that was previously opened, then waits for a reply from that process.

Waited I/O read operation

If a waited I/O **PUT_FILE_WRITEREAD_( )** call is executed, the *count_read* parameter indicates the number of bytes actually read.

Nowait I/O read operation

If a nowait I/O **PUT_FILE_WRITEREAD_( )** call is executed, *count_read* has no meaning and can be omitted. The count of the number of bytes read is obtained when the I/O operation completes through the *count-transferred* parameter of the Guardian AWAITIOX procedure or FILE_COMPLETE_ procedure.

The **PUT_FILE_WRITEREAD_( )** function must complete with a corresponding call to the Guardian AWAITIOX procedure or FILE_COMPLETE procedure when used with a file that is opened for nowait I/O.

Do not change the contents of the data buffers between the initiation and completion of a nowait **PUT_FILE_WRITEREAD_( )** operation. A retry can copy the data again from the user buffer and cause the wrong data to be written. Avoid sharing a buffer between a **PUT_FILE_WRITEREAD_( )** and another I/O operation because the contents of the data buffer might change before the write is completed.

Location of *write_buffer*, *read_buffer*, and *count_read*

The buffers and count transferred can be in the user stack or in an extended data segment. The *write_buffer*, *read_buffer*, and *count_read* cannot be in the user code space.

If the *write_buffer*, *read_buffer*, and *count_read* are in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Use on files opened for nowait I/O

- If the buffers are in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or FILE_COMPLETE procedure or is canceled by a call to the **PUT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- The extended data segment containing the buffers should not be in use at the time of the call to AWAITIOX or FILE_COMPLETE procedure.
- You can call **PUT_CANCEL( )** or the Guardian CANCELREQ to cancel nowait I/O initiated with **PUT_FILE_WRITEREAD_( )**. The I/O is canceled if the file is closed before the I/O completes or if you call the Guardian AWAITIOX procedure or FILE_COMPLETE procedure with a positive time limit and specific file number and the request times out.

Bounds checking

If the extended address of *write_buffer* or *read_buffer* is odd, bounds checking rounds the address to the next lower word boundary and also checks an extra byte. The odd address is used for the transfer.

**RETURN VALUES**

The **PUT_FILE_WRITEREAD_( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

**ERRORS**

None. This function does not set the **errno** variable.

**RELATED INFORMATION**

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**put_generateTag -** Increments and returns a static long tag

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**long put_generateTag(void);**

## PARAMETERS

None.

## DESCRIPTION

Increments and returns a static long string appropriate for use as a tag. Note that this long string will eventually wrap, thereby returning tags that may still be in use. For example, if a process calls **put_generateTag( )** 100 times per second, every second, the wrap will occur on the 248th day.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

This function returns a long tag.

## NAME

**put_getTMFConcurrentTransactions -** Gets the number of concurrent TMF transactions being used

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**int put_getTMFConcurrentTransactions(void);**

## PARAMETERS

None.

## DESCRIPTION

This function gets the number of concurrent TMF transactions being used.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

Upon successful completion, this function returns as an integer value the number of transactions being used.

## RELATED INFORMATION

Functions: **put_setTMFConcurrentTransactions(2)**.

## NAME

**put_INITRECEIVE -** Registers $RECEIVE filename

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**long put_INITRECEIVE(**
    **const short** *filenum***,**
    **const short** *receive_depth***);**

## PARAMETERS

*filenum* Specifies Guardian file number whose IO has completed.

*receive_depth* Specifies the maximum number of incoming messages as specified in the *filenum* value is **FILE_OPEN( )** call.

## DESCRIPTION

This function registers *filenum* as being managed by the $RECEIVE callback.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

This function returns Guardian error numbers, which include:

0 $RECEIVE was successfully registered.

| | |
|---|---|
| 29 | $RECEIVE was already registered prior to this call. |
| 29 | **FILE_COMPLETE_SET_( )** addition of $RECEIVE returned nonzero. |
| 29 | Value for *filenum* not 0. |

## NAME

**put_INITRECEIVEL -** Registers $RECEIVE filename (larger message version)

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**long put_INITRECEIVEL(**
    **const short** *filenum***,**
    **const short** *receive_depth***);**

## PARAMETERS

*filenum* Specifies Guardian file number whose IO has completed.

*receive_depth* Specifies the maximum number of incoming messages as specified in the *filenum* value is **FILE_OPEN( )** call.

## DESCRIPTION

This function is the same as the **put_INITRECEIVE( )** function, except:

- This function can handle the longer message lengths allowed by the **PUT_SERVERCLASS_SENDL_( )** function.
- The Guardian file-system error 4184 (EVERSION) can be returned.

See the **put_INITRECEIVE(2)** reference page.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## NOTES

This function is supported on systems running J06.10 and later J-series RVUs and H06.21 and later H-series RVUs, and must be used instead of the **put_INITRECEIVE( )** function when the messages are larger than 32 kilobytes. This function also can be used for shorter messages.

## RETURN VALUES

See the **put_INITRECEIVE(2)** reference page.

In addition, this function can return this Guardian file-system error:

4184 (EVERSION)
: The function was called from a system that is running a J-series RVU earlier than J06.10 or an H-series RVU earlier than H06.21.

## RELATED INFORMATION

Functions: **put_INITRECEIVE(2)**, **PUT_SERVERCLASS_SENDL_(3)**.

## NAME

**put_interrupt -** Interrupts all threads awaiting input or output

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**put_error_t put_interrupt(**
    **const short** *filenum***,**
    **const put_error_t** *errorPUT***);**

## PARAMETERS

*filenum* Specifies the Guardian file number for the file whose awaiting I/O is to be interrupted.

*errorPUT* Specifies PUT error returned to waiting file.

## DESCRIPTION

Interrupts all threads awaiting IO on file number. Note the I/O is not cancelled by this function. Interrupted threads will return from the **put_awaitio( )** function with a return value of *error_PUT*. Additionally, the *error* parameter passed to the **put_awaitio( )** function will be set as shown in the **PARAMETERS** section.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

**PUT_SUCCESS**
The file number awaiting I/O (if any) was interrupted.

**PUT_ERROR** Either the value specified for *error_PUT* is invalid or the value for *filenum* is less than 0 (zero) or is not registered.

## ERRORS

-1 **- PUT_ERROR**

40 **- PUT_TIMEOUT**

[EINTR] **- PUT_INTERRUPTED**

## NAME

**put_interruptTag -** Interrupts thread awaiting tagged I/O

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**put_error_t put_interruptTag(**
  **const short** *filenum***,**
  **const long** *tag***,**
  **const put_error_t** *error_PUT***);**

## PARAMETERS

*filenum* Specifies the Guardian file number for the file whose awaiting I/O is to be interrupted.

*tag* Specifies tag whose awaiting I/O is to be interrupted.

*error_PUT* Specifies PUT error returned to awaiting IO.

## DESCRIPTION

Interrupts the thread awaiting the tagged I/O on file number. Note that the I/O is not cancelled by this function. Interrupted threads will return from the **put_awaitio( )** function with a return value of *error_PUT*. Additionally, the *error* parameter passed to **put_awaitio( )** will be set as shown in the **ERRORS** section.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

**PUT_SUCCESS**
Awaiting IO was interrupted.

**PUT_ERROR** One of the following conditions exists:

- The value of *filenum* was less than 0 (zero), or no awaiting I/O was found
- The value of *filenum* is not registered
- The value for *error_PUT* is invalid

## ERRORS

| | |
|---|---|
| -1 | **PUT_ERROR** |
| 40 | **PUT_TIMEDOUT** |
| EINTR | **PUT_INTERRUPTED** |

## NAME

**PUT_LOCKFILE -** Excludes other users from accessing a Guardian disk file

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_LOCKFILE(
        short filenum,
        long tag);
```

## PARAMETERS

*filenum* Specifies the file number of a Guardian disk file open instance that identifies the file to be locked.

*tag* (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **PUT_LOCKFILE( )** function is a thread-aware version of the Guardian LOCKFILE procedure.

The **PUT_LOCKFILE( )** function is used to exclude other users from accessing a file (and any records within that file). The user is defined either as the opener of the file (identified by filenum) if the file is not audited or as the transaction (identified by the TRANSID) if the file is audited. If the file is currently unlocked or is locked by the current user when **PUT_LOCKFILE( )** is called, the file (and all its records) becomes locked, and the caller continues executing. If the file is already locked by another user, the behavior of the system is specified by the locking mode. Two locking modes are available:

Default The process requesting the lock is suspended. See the **Considerations** subsection of this reference page.

Alternate The lock request is rejected with Guardian file-system error 73. When the alternate locking mode is in effect, the process requesting the lock is not suspended. See the **Considerations** subsection of this reference page.

For programming information about the LOCKFILE procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## Considerations

Record locking versus file locking

A call to **PUT_LOCKFILE( )** is not equivalent to locking all records in a file; that is, locking all records still allows insertion of new records, but file locking does not. File locks and record locks are queued in the order in which they are issued.

Nowait and **PUT_LOCKFILE( )**

If the **PUT_LOCKFILE( )** function is used to initiate an operation with a file opened for nowait I/O, it must complete with a corresponding call to the Guardian AWAITIO procedure.

Locking modes

Default mode: If the file is already locked by another user when **PUT_LOCKFILE( )** is called, the process requesting the lock is suspended and queued in a locking queue behind other users trying to access the file. When the file becomes unlocked, the user at the head of the locking queue is granted access to the file. If the user at the head of the locking queue is requesting a lock, the user is granted the lock and resumes execution. If the user at the head of the locking queue is requesting a read, the read operation continues to completion.

Alternate mode: If the file is already locked by another user when the call to **PUT_LOCKFILE( )** is made, the lock request is rejected, and the call to **PUT_LOCKFILE( )** completes immediately with Guardian file-system error 73 (`file is locked`). The alternate locking mode is specified by calling the **PUT_SETMODE( )** procedure and specifying function 4.

Locks and open files (applies to nonaudited files only)

Locks are granted on a file open (that is, on a file number) basis. Therefore, if a process has multiple opens of the same file, a lock of one file number excludes access to the file through other file numbers.

Attempting to read a locked file in default locking mode

If the default locking mode is in effect when a call to **PUT_READX( )** or **PUT_READUPDATEX( )** is made for a file that is locked by another user, the caller of **PUT_READX( )** or **PUT_READUPDATEX( )** is suspended and queued in the locking queue behind other users attempting to access the file.

For nonaudited files, a deadlock condition (a permanent suspension of your application) occurs if **PUT_READX( )** or **PUT_READUPDATEX( )** is called by the process that has a record locked with a file number other than that supplied in the **PUT_READX( )** or **PUT_READUPDATEX( )** call. For an explanation of multiple opens by the same process, see the **PUT_FILE_OPEN_(2)** reference page either online or in the *Open System Services System Calls Reference Manual*.

Accessing a locked file

If the file is locked by a user other than the caller at the time of the call, the call is rejected with Guardian file-system error 73 (`file is locked`) when:

**PUT_READX( )** or **PUT_READUPDATEX( )** is called, and the alternate locking mode is in effect.

**PUT_WRITEX( )**, WRITEUPDATE, or **PUT_CONTROL( )** is called.

A count of the locks in effect is not maintained. Multiple locks can be unlocked with one call to **PUT_UNLOCKFILE( )**.

### Use on OSS Objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **PUT_LOCKFILE( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**PUT_LOCKREC -** Excludes other users from accessing a record in a Guardian disk file

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <pthread.h>**

**short PUT_LOCKREC(**
  **short** *filenum***,**
  **long** *tag***);**

## PARAMETERS

*filenum* Specifies the file number of a Guardian disk file open instance that identifies the file containing the record to be locked.

*tag* (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **PUT_LOCKREC( )** function is a thread-aware version of the Guardian LOCKREC procedure.

The LOCKREC procedure excludes other users from accessing a record at the current position. The user is defined either as the opener of the file (identified by *filenum*) if the file is not audited or as the transaction (identified by the TRANSID) if the file is audited.

For key-sequenced, relative, and entry-sequenced files, the current position is the record with a key value that matches exactly the current key value. For unstructured files, the current position is the relative byte address (RBA) identified by the current-record pointer. If the record is unlocked when **PUT_LOCKREC( )** is called, the record becomes locked, and the caller continues executing.

You cannot use **PUT_LOCKREC( )** with queue files.

If the file is already locked by another user, the behavior of the system is specified by the locking mode. Two locking modes are available:

Default The process requesting the lock is suspended. See the **Considerations** subsection of this reference page.

Alternate The lock request is rejected with Guardian file-system error 73. When the alternate locking mode is in effect, the process requesting the lock is not suspended. See the **Considerations** subsection of this reference page.

For programming information about the LOCKREC procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

### Considerations

Record locking versus file locking

A call to **PUT_LOCKFILE()** is not equivalent to locking all records in a file; that is, locking all records still allows insertion of new records, but file locking does not. File locks and record locks are queued in the order in which they are issued.

Nowait and **PUT_LOCKREC()**

If the **PUT_LOCKREC()** function is used to initiate an operation with a file opened for nowait I/O, it must complete with a corresponding call to the Guardian AWAITIO procedure.

Default locking mode

If the record is already locked by another user when **PUT_LOCKREC()** is called, the process requesting the lock is suspended and queued in a locking queue behind other users also requesting to lock or read the record.

When the record becomes unlocked, the user at the head of the locking queue is granted access to the record. If the user at the head of the locking queue is requesting a lock, it is granted the lock and resumes execution. If the user at the head of the locking queue is requesting a read operation, the read operation continues to completion.

Alternate locking mode

If the record is already locked by another user when **PUT_LOCKREC()** is called, the lock request is rejected, and the call to **PUT_LOCKREC()** completes immediately with Guardian file-system error 73 (`record is locked`). The alternate locking mode is specified by calling the **PUT_SETMODE()** procedure and specifying function 4.

Attempting to read a locked record in default locking mode

If the default locking mode is in effect when **PUT_READX()** or **PUT_READUPDATEX()** is called for a record that is locked by another user, the caller to **PUT_READX()** or **PUT_READUPDATEX()** is suspended and queued in the locking queue behind other users attempting to lock or read the record. (Another user means another open *filenum* if the file is not audited, or another TRANSID if the file is audited.)

For nonaudited files, a deadlock condition (a permanent suspension of your application) occurs if **PUT_READX()** or **PUT_READUPDATEX()** is called by the process that has a record locked with a file number other than that supplied in the **PUT_READX()** or **PUT_READUPDATEX()** call. For an explanation of multiple opens by the same process, see the **PUT_FILE_OPEN_(2)** reference page either online or in the *Open System Services System Calls Reference Manual*.

Selecting the locking mode with **PUT_SETMODE()**

The locking mode is specified by the calling SETMODE procedure with function 4.

A count of the locks in effect is not maintained. Multiple locks can be unlocked with one call to **PUT_UNLOCKFILE()**.

Structured files

Calling LOCKREC after positioning on a nonunique key

If the call to **PUT_LOCKREC()** immediately follows a call to KEYPOSITION where a nonunique alternate key is specified, the call to **PUT_LOCKREC()** fails. A subsequent call to the Guardian FILE_GETINFO_ or FILEINFO procedure shows that a Guardian file-system error 46 (`invalid key`) occurred. However, if an intermediate call to **PUT_READX()** is performed, the call to **PUT_LOCKREC()** is permitted because a unique record is identified.

Current-state indicators after **PUT_LOCKREC()**

After a successful call to **PUT_LOCKREC()**, current-state indicators are unchanged.

Unstructured files

Locking the relative byte address (RBA) in an unstructured file

Record positions in an unstructured file are represented by an RBA, and the RBA can be locked with **PUT_LOCKREC()**. To lock a position in an unstructured file, first call the Guardian POSITION procedure with the desired RBA, and then call **PUT_LOCKREC()**. This locks the RBA; any other process attempting to access the file with exactly the same RBA encounters a `record is locked` condition. You can access that RBA by positioning to RBA-2. Depending on the process's locking mode, the call either fails with Guardian file-system error 73 (`record is locked`) or is placed in the locking queue.

Record pointers after a call to **PUT_LOCKREC( )**

After a call to **PUT_LOCKREC( )**, the current-record, next-record, and end-of-file pointers remain unchanged.

Ways to avoid or resolve deadlocks

One way to avoid deadlock is to call function 4 of the **PUT_SETMODE( )** procedure to establish one of the alternate locking modes. A common method of avoiding deadlock situations is to lock records in some predetermined order. Deadlocks can be resolved if you lock records using a nowait open and call the Guardian AWAITIO procedure with a timeout specified.

### Use on OSS Objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **PUT_LOCKREC( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**PUT_READLOCKX -** Sequentially locks and reads records in a Guardian disk file

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_READLOCKX(
        short filenum,
        char *buffer,
       #ifdef __LP64
        int read_count,
        int *count_read,
       #else
        unsigned short read_count,
        unsigned short *count_read,
       #endif
        long tag);
```

## PARAMETERS

### Input

*filenum* Specifies the file number of a Guardian file open instance that identifies the file to be read.

*read_count* Specifies the number of bytes to be read.

*tag* (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

### Output

*buffer* Specifies an array in the application process in which the information read from the file is returned.

*count_read* (Optional) For waited I/O only. This parameter returns a count of the number of bytes returned from the file into *buffer*.

## DESCRIPTION

The **PUT_READLOCKX( )** function is a thread-aware version of the Guardian READLOCKX procedure.

The **PUT_ READLOCKX( )** function sequentially locks and reads records in a Guardian disk file, exactly like the combination of a **PUT_LOCKREC( )** and **PUT_READX( )** call.

For programming information about the READLOCKX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

### Considerations

Buffer use
: **PUT_READLOCKX( )** is intended for use with 32-bit extended addresses and 64-bit extended addresses. The data buffer for **PUT_READLOCKX( )** can be either in the caller's stack segment or any extended data segment.

Nowait I/O and **PUT_READLOCKX( )**
: If the **PUT_READLOCKX( )** function is used to initiate an operation with a file opened for nowait I/O, it must complete with a corresponding call to the Guardian AWAITIOX procedure.

Use for key-sequenced, relative, and entry-sequenced files
: For key-sequenced, relative, and entry-sequenced files, a subset of the file (defined by the current access path, positioning mode, and comparison length) is locked and read with successive calls to **PUT_READLOCKX( )**.

: For key-sequenced, relative, and entry-sequenced files, the first call to **PUT_READLOCKX( )** after a positioning (or open) locks and then returns the first record of the subset. Subsequent calls to **PUT_READLOCKX( )** without intermediate positioning locks returns successive records in the subset. After each of the subset's records are read, the position of the record just read becomes the file's current position. An attempt to read a record following the last record in a subset returns an EOF indication.

Locking records in an unstructured file
: You can use **PUT_READLOCKX( )** to lock record positions, represented by a relative byte address (RBA), in an unstructured file. When sequentially reading an unstructured file with **PUT_READLOCKX( )**, each call to **PUT_READLOCK[X( )** first locks the RBA stored in the current next-record pointer and then returns record data beginning at that pointer for *read_count* bytes. After a successful call to **PUT_READLOCK[X( )**, the current-record pointer is set to the previous next-record pointer, and the next-record pointer is set to the previous next-record pointer plus *read_count*. This process repeats for each subsequent call to **PUT_READLOCKX( )**.

Location of *buffer* and *count_read*
The buffer and count transferred can be in the user stack or in an extended data segment. The *buffer* and *count_read* cannot be in the user code space.

If the *buffer* and *count_read* is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Transfer size The size of the transfer is subject to current restrictions for the type of file.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **PUT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This restriction also applies to other processes that might be sharing the segment. It is the application's responsibility to ensure this.
- If you initiated the I/O with **PUT_READLOCKX( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- A selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- You can cancel Nowait I/O initiated with **PUT_READLOCKX( )** with a call to **PUT_CANCEL( )** or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or if the Guardian AWAITIOX procedure is called with a positive time limit and specific file number and the request times out.

Use of buffers A file opened by **PUT_FILE_OPEN_( )** uses direct I/O transfers by default; you can use **PUT_SETMODE(72)** to force the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers.

Bounds checking
If the extended address of *buffer* is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

All considerations for the **PUT_READX( )** function also apply to this function.

### Use on OSS objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **PUT_READLOCKX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors* *and Messages Manual*.

**ERRORS**

None. This function does not set the **errno** variable.

**RELATED INFORMATION**

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**PUT_READUPDATELOCKX -** Allows random processing of records in a Guardian disk file

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_READUPDATELOCKX(
        short filenum,
        char *buffer,
       #ifdef __LP64
        int read_count,
        int *count_read,
       #else
        unsigned short read_count,
        unsigned short *count_read,
       #endif
        long tag);
```

## PARAMETERS

### Input

*filenum*
: Specifies the file number of a Guardian file open instance that identifies the file to be read.

*read_count*
: Specifies the number of bytes to be read.

*tag*
: (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

  This parameter is supported only for program compatibility; if you provide it, it is ignored.

### Output

*buffer*
: Specifies an array in the application process in which the information read from the file is returned.

*count_read*
: (Optional) For waited I/O only. This parameter returns a count of the number of bytes returned from the file into *buffer*.

## DESCRIPTION

The **PUT_READUPDATELOCKX( )** function is a thread-aware version of the Guardian READUPDATELOCKX procedure.

You use **PUT_READUPDATELOCKX( )** function for random processing of records in a Guardian disk file. This function first locks then reads the record from the current position in the file in anticipation of a subsequent call to the **PUT_WRITEUPDATEX( )** or **PUT_WRITEUPDATEUNLOCK( )** procedure. **PUT_READUPDATELOCKX( )** is intended for reading a record after calling the Guardian POSITION or KEYPOSITION procedure.

**PUT_READUPDATELOCKX( )** locks and reads the record in the same manner as the combination of the Guardian LOCKREC and READUPDATEX procedures but requires less system processing than the two separate calls would require.

For programming information about the READUPDATELOCKX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## Considerations

Buffer use
: **PUT_READUPDATELOCKX( )** is intended for use with 32-bit extended addresses and 64-bit extended addresses. The data buffer for **PUT_READUPDATELOCKX( )** can be either in the caller's stack segment or any extended data segment.

Nowait I/O and **PUT_READUPDATELOCKX( )**
: The **PUT_READUPDATELOCKX( )** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

Use on nondisk files
: If **PUT_READUPDATELOCKX( )** is performed on nondisk files, an error is returned.

Random processing
: For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. Therefore, positioning for **PUT_READUPDATELOCKX( )** is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to **PUT_READUPDATELOCKX( )** is rejected with Guardian file-system error 11.

Queue files
: To use **PUT_READUPDATELOCKX( )**, you must open a queue file with write access and with a *sync_or_receive_depth* of 0 (zero).

Location of *buffer* and *count_read*

The buffer and count transferred can be in the user stack or in an extended data segment. The *buffer* and *count_read* cannot be in the user code space.

If the *buffer* and *count_read* is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **PUT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This restriction also applies to other processes that might be sharing the segment. It is the application's responsibility to ensure this.
- If you initiated the I/O with **PUT_READUPDATELOCKX( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- A selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- You can cancel nowait I/O initiated with **PUT_READUPDATELOCKX( )** with a call to **PUT_CANCEL( )** or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or if the Guardian AWAITIOX procedure is called with a positive time limit and specific file number and the request times out.

Use of buffers

A file opened by **PUT_FILE_OPEN_( )** uses direct I/O transfers by default; you can use **PUT_SETMODE(72)** to force the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers.

Bounds checking

If the extended address of *buffer* is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

All considerations for the **PUT_LOCKREC( )** function also apply to this function. See also the "Disk File Considerations" for the Guardian READUPDATE procedure.

### Use on OSS objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **PUT_READUPDATELOCKX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

**ERRORS**

None. This function does not set the **errno** variable.

**RELATED INFORMATION**

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**PUT_READUPDATEX -** Reads data from a Guardian disk or process file in anticipation of a subsequent write to the file

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_READUPDATEX(
        short filenum,
        char *buffer,
       #ifdef __LP64
        int read_count,
        int *count_read,
       #else
        unsigned short read_count,
        unsigned short *count_read,
       #endif
        long tag);
```

## PARAMETERS

### Input

*filenum* Specifies the file number of a Guardian file open instance that identifies the file to be read.

*read_count* Specifies the number of bytes to be read.

*tag* (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

### Output

*buffer* Specifies an array in the application process in which the information read from the file is returned.

*count_read* (Optional) For waited I/O only. This parameter returns a count of the number of bytes returned from the file into *buffer*.

## DESCRIPTION

The **PUT_READUPDATEX( )** function is a thread-aware version of the Guardian READUPDATEX procedure.

This function reads data from a disk or process file in anticipation of a subsequent write to the file. The values of the current-record and next-record pointers do not change. This function has the following uses:

Disk files **PUT_READUPDATEX( )** is used for random processing. Data is read from the file at the position of the current-record pointer. A call to this function typically follows a corresponding call to the Guardian POSITION or KEYPOSITION procedure.

Queue Files
**PUT_READUPDATEX( )** is not supported on queue files. An attempt to use **PUT_READUPDATEX( )** is rejected with Guardian file-system error 2.

Interprocess communication
**PUT_READUPDATEX( )** reads a message from the $RECEIVE file that is answered in a later call to the Guardian REPLYX procedure. Each message read by **PUT_READUPDATEX( )** must be replied to in a corresponding call to REPLYX.

For programming information about the READUPDATEX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## Considerations

Buffer use
**PUT_READUPDATEX( )** is intended for use with 32-bit extended addresses and 64-bit extended addresses. The data buffer for **PUT_READUPDATEX( )** can be either in the caller's stack segment or any extended data segment.

Random processing and positioning
A call to **PUT_READUPDATEX( )** returns the record from the current position in the file. Because **PUT_READUPDATEX( )** is designed for random processing, it cannot be used for successive positioning through a subset of records as the **PUT_READX( )** function does. Rather, **PUT_READUPDATEX( )** reads a record after a call to the Guardian POSITION or KEYPOSITION procedure, in anticipation of a subsequent update through a call to the Guardian WRITEUPDATEX procedure.

Calling **PUT_READUPDATEX( )** after **PUT_READX( )**

A call to **PUT_READUPDATEX( )** after a call to **PUT_READX( )**, without intermediate positioning, returns the same record as the call to **PUT_READX( )**.

Waited **PUT_READUPDATEX( )**

If a waited **PUT_READUPDATEX( )** call is executed, the *count_read* parameter indicates the number of bytes actually read.

Nowait I/O and **PUT_READUPDATEX( )**

If a nowait **PUT_READUPDATEX( )** call is executed, *count_read* has no meaning and can be omitted. The count of the number of bytes read is obtained when the I/O operation completes through the *count_transferred* parameter of the Guardian AWAITIOX procedure. The **PUT_READUPDATEX( )** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

Default locking mode action

If the default locking mode is in effect when a call to **PUT_READUPDATEX( )** is made to a locked file or record, but the *filenum* of the locked file differs from the *filenum* in the call, the caller of **PUT_READUPDATEX( )** is suspended and queued in the locking queue behind other processes attempting to access the file or record.

### Use on OSS objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **PUT_READUPDATEX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**PUT_READX -** Returns data from an open Guardian file to the application process data area

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_READX(
        short filenum,
        char *buffer,
       #ifdef __LP64
        int read_count,
        int *count_read,
       #else
        unsigned short read_count,
        unsigned short *count_read,
       #endif
        long tag);
```

## PARAMETERS

### Input

*filenum* Specifies the file number of a Guardian file open instance that identifies the file to be read.

*read_count* (Optional) Specifies the number of bytes to be read.

*tag* (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

### Output

*buffer* Specifies an array in the application process in which the information read from the file is returned.

*count_read* (Optional) For waited I/O only. This parameter returns a count of the number of bytes returned from the file into *buffer*.

## DESCRIPTION

The **PUT_READX( )** function is a thread-aware version of the Guardian READX procedure.

The **PUT_READX( )** function returns data from an open Guardian file to the application process's data area. The **PUT_READX( )** function sequentially reads a disk file. For key-sequenced, relative, and entry-sequenced files, the **PUT_READX( )** function reads a subset of records in the file. (A subset of records is defined by an access path, positioning mode, and comparison length.)

For programming information about the Guardian READX file-system procedure, see the *Guardian Programmer's Guide*, the *Enscribe Programmer's Guide*, and the manuals for your specific data communications interface.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**General Considerations**

Buffer use
: **PUT_READX( )** is intended for use with 32-bit extended addresses and 64-bit extended addresses. The data buffer for **PUT_READX( )** can be either in the caller's stack segment or any extended data segment.

Waited **PUT_READX( )**
: If a waited **PUT_READX( )** call is executed, the *count_read* parameter indicates the number of bytes actually read.

Nowait **PUT_READX( )**
: If a nowait **PUT_READX( )** call is executed, *count_read* has no meaning and can be omitted. The count of the number of bytes read is obtained through the *count-transferred* parameter of the Guardian AWAITIOX procedure when the I/O operation completes.

: The **PUT_READX( )** function must complete with a call to the Guardian AWAITIOX procedure when it is used with a file that is opened for nowait I/O.

: It is possible to initiate concurrent nowait read operations that share the same data buffer. To do this successfully with files opened by **PUT_FILE_OPEN_( )**, you must use **PUT_SETMODE( )** function 72 to cause the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers.

**PUT_READX( )** call when default locking mode is in effect
: If the default locking mode is in effect when a call to **PUT_READX( )** is made to a locked file, but the *filenum* of the locked file differs from the *filenum* in the call, the caller of **PUT_READX( )** is suspended and queued in the locking queue behind other processes attempting to lock or read the file or record.

: A deadlock condition occurs if a call to **PUT_READX( )** is made by a process having multiple opens on the same file and the *filenum* used to lock the file differs from the *filenum* supplied to **PUT_READX( )**.

Read call when alternate locking mode is in effect

If the alternate locking mode is in effect when **PUT_READX( )** is called, and the file or record is locked through a Guardian file number other than that supplied in the call, the call is rejected with Guardian file-system error 73 (`file is locked`).

Locking mode for read

The locking mode is specified by **PUT_SETMODE( )** function 4. If you encounter Guardian file-system error 73 (`file is locked`), you do not need to call **PUT_SETMODE( )** for every call to **PUT_READX( )**. **PUT_SETMODE() )** stays in effect indefinitely (for example, until another **PUT_SETMODE( )** call is performed or the file is closed), and no additional overhead is involved.

Location of *buffer* and *count_read*

The buffer and count transferred can be in the user stack or in an extended data segment. The *buffer* and *count_read* cannot be in the user code space.

If the *buffer* and *count_read* are in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **PUT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This restriction also applies to other processes that might be sharing the segment. It is the application's responsibility to ensure this.
- If the I/O has been initiated with **PUT_READX( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- A selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- You can cancel nowait I/O initiated with **PUT_READX( )** with a call to **PUT_CANCEL( )** or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or if the Guardian AWAITIOX procedure is called with a positive time limit and specific file number and the request times out.

Use of buffers

A file opened by **PUT_FILE_OPEN_( )** uses direct I/O transfers by default; you can use **PUT_SETMODE(72)** to force the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers.

Bounds checking

If the extended address of *buffer* is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

Queue files

You can use **PUT_READX( )** to perform a nondestructive read of a queue file record. If the Guardian KEYPOSITIONX procedure is used to position to the beginning of the file, the first **PUT_READX( )** call performed returns a record with a length of 8 bytes and contents of all zeros. Subsequent **PUT_READX( )** calls return data from records written to the file.

**Disk File Considerations**

Large data transfers for unstructured files using default mode

For all read procedures, using default mode allows I/O sizes for unstructured files to be as large as 56 kilobytes (57,344), if the unstructured buffer size is 4 KB (4096). Default mode here refers to the mode of the file if **PUT_SETMODE( )** function 141 is not invoked.

For an unstructured file with an unstructured buffer size other than 4 KB, DP2 automatically adjusts the unstructured buffer size to 4 KB, if possible, when an I/O larger than 4KB is attempted. However, this adjustment is not possible for files that have extents with an odd number of pages; in such cases an I/O over 4 KB is not possible. The switch to a different unstructured buffer size will have a transient performance impact, so HP recommends that you set the size 4 KB initially, which is the default. Transfer sizes over 4 KB are not supported in default mode for unstructured access to structured files.

Large data transfers using **PUT_SETMODE(141)**

For **PUT_READX( )** only, large data transfers (more than 4096 bytes) can be done for unstructured access to structured or unstructured files, regardless of unstructured buffer size, by using **PUT_SETMODE( )** function 141. When you use **PUT_SETMODE(141)** to enable large data transfers, you can specify up to 56K (57344) bytes for the *read_count* parameter. For an explanation of function 141, see the Guardian SETMODE procedure description in the *Guardian Procedure Calls Reference Manual*.

Structured files

A subset of records for sequential **PUT_READX( )** calls

The subset of records read by a series of calls to **PUT_READX( )** is specified through calls to the Guardian POSITION or KEYPOSITION procedures.

Reading of an approximate subset of records

If an approximate subset is being read, the first record returned is the one whose key field, as indicated by the current key specifier, contains a value equal to or greater than the current key. Subsequent reading of the subset returns successive records until the last record in the file is read (an EOF indication is then returned).

Reading of a generic subset of records

If a generic subset is being read, the first record returned is the one whose key field, as designated by the current-key specifier, contains a value equal to the current key for *comparison-length* bytes. Subsequent reading of the file returns successive records whose key matches the current key (for *comparison-length* bytes). When the current key no longer matches, an EOF indication returns.

For relative and entry-sequenced files, a generic subset of the primary key is equivalent to an exact subset.

Reading of an exact subset of records

If an exact subset is being read, the only records returned are those whose key field, as designated by the current-key specifier, contains a value of exactly the *comparison length* bytes (see the Guardian KEYPOSITION procedure in the *Guardian Procedure Calls Reference Manual*) and is equal to the key. When the current key no longer matches, an EOF indication returns. The exact subset for a key field having a unique value is at most one record.

Indicators after **PUT_READX( )** call

After a successful **PUT_READX( )** call, the current-state indicators have these values:

- Current position is the record just read.
- Positioning mode is unchanged.
- Comparison length is unchanged.
- Current primary-key value is set to the value of the primary-key field in the record.

Unstructured files

Data transfer

Data transfer begins from an unstructured disk file at the position indicated by the next-record pointer. The READ[X] procedure reads records sequentially on the basis of a beginning relative byte address (RBA) and the length of the records read.

Odd unstructured

If the unstructured file is created with the odd unstructured attribute (also known as ODDUNSTR) set, the number of bytes read is exactly the number of bytes specified with *read_count*. If the odd unstructured attribute is not set when the file is created, the value of *read_count* is rounded up to an even number before the **PUT_READX( )** operation is executed.

You set the odd unstructured attribute with the Guardian FILE_CREATE_, FILE_CREATELIST_, or CREATE procedure, or with the File Utility Program (FUP) SET and CREATE commands.

*read_count*

Unstructured files are transparently blocked. The BUFFERSIZE file attribute value, if not set by the user, defaults to 4096 bytes. The BUFFERSIZE attribute value (which is set by specifying **PUT_SETMODE( )** function 93) does not constrain the allowable *read_count* in any way. However, there is a performance penalty if the **PUT_READX( )** call does not start on a BUFFERSIZE boundary and does not have a *read_count* that is an integral multiple of the BUFFERSIZE. The DP2 disk process executes your requested I/O in (possibly multiple) units of BUFFERSIZE blocks starting on a block boundary.

*count_read* for unstructured reads

After a successful call to **PUT_READX( )** for an unstructured file, the value returned in *count_read* is the minimum of *read_count* or the EOF pointer minus the next-record pointer.

Pointers after **PUT_READX( )** call

After a successful **PUT_READX( )** call to an unstructured file, the file pointers are:

- Current-record pointer is old next-record pointer.
- Next-record pointer is old next-record pointer plus *count_read*.

## RETURN VALUES

The **PUT_READX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

**NAME**

**put_RECEIVEREAD -** Initiates thread-aware function for reading $RECEIVE

**LIBRARY**

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

**SYNOPSIS**

```
#include <pthread.h>

long put_RECEIVEREAD(
        const short filenum,
        char *buffer,
        const short read_count,
        int *count_read,
        const long timelimit,
        short *receive_info,
        short *dialog_info);
```

**PARAMETERS**

*filenum* Specifies the Guardian file number for $RECEIVE (always 0).

*buffer* Specifies the data buffer.

*read_count* Specifies the number of bytes to read.

*count_read* Specifies the number of bytes read.

*timelimit* Specifies a FILE_COMPLETE-style time limit.

*receive_info* Specifies a FILE_GETRECEIVEINFO-style $RECEIVE info structure; NULL may be passed if this information is not needed; must not be NULL if *filenum*'s *receive_depth* is greater than 0 (zero).

*dialog_info* Specifies a FILE_GETRECEIVEINFO-style of dialog information (a **short int** used by context-sensitive Pathway servers); NULL can be passed if this information is not needed; NULL must be passed if *receive_info* is NULL.

**DESCRIPTION**

This thread-aware function is specifically for reading $RECEIVE. **put_RECEIVEREAD( )** is slightly patterned after a combination of the READUPDATEX procedure and the FILE_GETRECEIVEINFO procedure, although its parameters do not match either of its modeled procedures. A side effect of calling **put_RECEIVEREAD )** puts the calling thread into a transaction (via a call to the **PUT_TMF_SetTxHandle( )** function), if the received message was transactional. The calling thread may be blocked to honor the *filenum* value's receive depth. This allows any number of threads to simultaneously call **put_RECEIVEREAD( )**. Blocked threads will be unblocked as other threads complete their calls to the **put_REPLYX( )** function.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## NOTES

Processing of the **put_RECEIVEREAD( )** function cannot be interrupted by specifying **put_interrupt(PUT_INTERRUPTED)**. The **put_RECEIVEREAD( )** function responds to the attempt by retrying the input or output.

To interrupt the **put_RECEIVEREAD( )** function, use one of the following function calls:

- **put_wakeup(0, -1, 0,** *error***)** where *error* is any error number that can be recognized as a return value for the **put_RECEIVEREAD( )** function.
- **put_interrupt(0, PUT_ERROR)**.
- **put_interrupt(0, PUT_TIMEDOUT)**.

Using any of these calls also cancels the input/output operation.

## RETURN VALUES

This function returns Guardian file-system error numbers including:

16 *filenum* is not registered.

## NAME

**put_RECEIVEREADL -** Initiates thread-aware function for reading $RECEIVE (larger message version)

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <pthread.h>

long put_RECEIVEREADL(
        const short filenum,
        char *buffer,
        const int read_count,
        int *count_read,
        const long timelimit,
        short *receive_info);
```

## PARAMETERS

*filenum* Specifies the Guardian file number for $RECEIVE (always 0).

*buffer* Specifies the data buffer.

*read_count* Specifies the number of bytes to read.

*count_read* Specifies the number of bytes read.

*timelimit* Specifies a FILE_COMPLETEL_-style time limit.

*receive_info* Specifies a FILE_GETRECEIVEINFOL_-style $RECEIVE info structure; NULL may be passed if this information is not needed; must not be NULL if *filenum*'s *receive_depth* is greater than 0 (zero).

## DESCRIPTION

This function is the same as the **put_RECEIVEREAD( )** function, except that:

- This function can handle the longer message lengths allowed by the **PUT_SERVERCLASS_SENDL_( )** function.
- The *read_count* parameter is type **const int**.
- The *dialog_info* parameter is not included in the **put_RECEIVEREADL( )** function.
- The Guardian file-system error 4184 (EVERSION) can be returned.

See the **put_RECEIVEREAD(2)** reference page.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**NOTES**

This function is supported on systems running J06.10 and later J-series RVUs and H06.21 and later H-series RVUs, and must be used instead of the **put_RECEIVEREAD( )** function when the messages are larger than 32 kilobytes. This function also can be used for shorter messages.

**RETURN VALUES**

See the **put_RECEIVEREAD(2)** reference page.

In addition, this function can return this Guardian file-system error:

4184 (EVERSION)
: The function was called from a system that is running a J-series RVU earlier than J06.10 or an H-series RVU earlier than H06.21.

**RELATED INFORMATION**

Functions: **put_RECEIVEREAD(2)**, **PUT_SERVERCLASS_SENDL_(3)**.

## NAME

**put_regFile -** Registers the file number

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**put_error_t put_regFile(**
**const short** *filenum***);**

## PARAMETERS

*filenum* Specifies the Guardian file number of the file being registered.

## DESCRIPTION

Registers the file number as one that the user will manage through the default callback.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

See the **put_regFileIOHandler(2)** reference page.

## NAME

**put_regFileIOHandler -** Registers the file number

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**put_error_t put_regFileIOHandler(**
**const short** *filenum***,**
**const put_FileIOHandler_p** *functionPtr***);**

## PARAMETERS

*filenum* Specifies the Guardian file number for the file being registered.

*functionPtr* Specifies user-supplied callback. This function must not block its invoking thread; for example, it should not call the **put_awaitio( )** function.

## DESCRIPTION

This function registers the file number as one that the user will manage through a user-supplied callback. This callback is invoked immediately after each I/O on *filenum* completes.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

**PUT_SUCCESS**

The Guardian file number was successfully registered.

**PUT_ERROR** The value specified for *filenum* was less than 0 (zero).

**PUT_ERROR** *filenum* was already registered prior to this call.

**PUT_ERROR** The FILE_COMPLETE_SET_ procedure addition of *filenum* returned a nonzero value.

**PUT_ERROR** *functionPtr* is NULL.

## NAME

**put_regOSSFileIOHandler -** Registers the file descriptor to manage through a callback function

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**put_error_t put_regOSSFileIOHandler(**
    **const int** *filedes***,**
    **const put_OSSFileIOHandler_p** *functionPtr***);**

## PARAMETERS

*filedes* Specifies the OSS file descriptor being registered.

*functionPtr* Specifies the user-supplied callback function; this function must not block.

## DESCRIPTION

This function registers the file descriptor as one that the user will manage through a user-supplied callback.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

**PUT_SUCCESS**
Value for file descriptor was registered.

**PUT_ERROR** The specified *filedes* was less than 0 (zero).

**PUT_ERROR** *filedes* was already registered prior to this call.

**PUT_ERROR** *functionPtr* is NULL.

## NAME

**put_regPathsendFile -** Registers the Pathsend file number

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**put_error_t put_regPathsendFile(**
  **const short** *fileno***);**

## PARAMETERS

*fileno* Contains the *scsend-op-num* value obtained during the first nowaited SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ procedure call.

## DESCRIPTION

This function is used to register the Pathsend file number. This function should be called immediately after the first call to a SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ procedure call.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

**PUT_SUCCESS**
The Pathsend file number was successfully registered.

**PUT_ERROR** The specified Pathsend file number is already registered.

## NAME

**put_regPathsendTagHandler** - Registers the user-supplied Pathsend tag

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**put_error_t put_regPathsendTagHandler(**
**const long** *tag***,**
**put_FileIOHandler_p** *callback***,**
**void *** *userdata* **);**

## PARAMETERS

*tag* Specifies the Pathsend tag that should be registered.

*callback* Specifies a user-supplied callback function. This function should not block its invoking thread. The callback function should have the following prototype:

```
callback(const short filenum,
                    /* Guardian file number
                      being waited on */
          const long tag,
                    /* tag being waited on or
                      -1 for all tags */
          const long completionCount,
                    /* byte transfer count
                      of completed IO */
          const long fserror,
                    /* Guardian error number for IO */
          void * userdata
                    /* for communication between
                      I/O initiator and callback. */
          );
```

*userdata* Specifies data to be communicated between the I/O initiator and the callback function.

## DESCRIPTION

This function registers the Pathsend tag as a tag that the user will manage through a user-supplied callback function. The callback function is invoked when a Pathsend operation that uses the tag completes.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

### RETURN VALUES

**PUT_SUCCESS**
The specified tag was registered.

**PUT_ERROR** Another Pathsend handler has already registered the tag.

### RELATED INFORMATION

Functions: **put_unregPathsendTagHandler(2)**, **PUT_SERVERCLASS_DIALOG_ABORT_(2)**, **PUT_SERVERCLASS_DIALOG_BEGIN_(2)**, **PUT_SERVERCLASS_DIALOG_END_(2)**, **PUT_SERVERCLASS_DIALOG_SEND_(2)**, **PUT_SERVERCLASS_SEND_INFO_(2)**, **PUT_SERVERCLASS_SEND_(2)**.

## NAME

**put_regTimerHandler -** Registers a user-supplied timer callback function

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**put_error_t put_regTimerHandler(**
**const put_TimerHandler_p** *functionPtr***);**

## PARAMETERS

*functionPtr* Specifies the user-supplied callback function; this function must not block I/O.

## DESCRIPTION

This function registers a user-supplied timer callback function.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

**PUT_SUCCESS**
The callback function was successfully registered.

**PUT_ERROR** *functionPtr* is NULL.

**PUT_ERROR** The specified callback function is already registered.

### NAME

**put_REPLYX -** Initiates thread-aware REPLYX procedure call

### LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

### SYNOPSIS

**#include <pthread.h>**

**long put_REPLYX(**
    **const char** ***buffer*****,**
    **const short** *write_count***,**
    **short** ***count_written*****,**
    **const short** *msg_tag***,**
    **const short** *error_return***);**

### PARAMETERS

*buffer* Specifies data buffer.

*write_count* Specifies the number of bytes to write.

*count_written* Specifies the number of bytes written; might be NULL.

*msg_tag* Specifies required tag identifying message to reply to and is ignored if the corresponding Guardian file number receive depth is 1.

*error_return* Specifies a Guardian file-system error to return to sender.

### DESCRIPTION

This is a thread-aware version of the REPLYX procedure call; this function clears the thread's transaction context if appropriate.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.

- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**RETURN VALUES**

This function returns a Guardian file-system error number.

## NAME

**put_REPLYXL -** Initiates thread-aware REPLYXL procedure call (larger message version)

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <pthread.h>

long put_REPLYXL(
        const char *buffer,
        const int write_countL,
        int *count_writtenL,
        const short msg_tag,
        const short error_return);
```

## PARAMETERS

*buffer* Specifies data buffer.

*write_countL* Specifies the number of bytes to write.

*count_writtenL* Specifies the number of bytes written; might be NULL.

*msg_tag* Specifies required tag identifying message to reply to and is ignored if the corresponding Guardian file number receive depth is 1.

*error_return* Specifies a Guardian file-system error to return to sender.

## DESCRIPTION

This function is the same as the **put_REPLYX( )** function, except:

- This function can handle the longer message lengths allowed by the **PUT_SERVERCLASS_SENDL_( )** function.
- The *write_countL* parameter is type **const int**.
- The *count_writtenL* parameter is type **int**.
- The Guardian file-system error 4184 (EVERSION) can be returned.

See the **put_REPLYX(2)** reference page.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**NOTES**

This function is supported on systems running J06.10 and later J-series RVUs and H06.21 and later H-series RVUs, and must be used instead of the **put_REPLYX( )** function when the messages are larger than 32 kilobytes long. This function also can be used for shorter messages.

**RETURN VALUES**

See the **put_REPLYX(2)** reference page.

In addition, this function can return this Guardian file-system error:

4184 (EVERSION)
: The function was called from a system that is running a J-series RVU earlier than to J06.10 or an H-series RVU earlier than H06.21.

## NAME

**put_select_single_np -** Initiates thread-aware **select( )** function for a single file descriptor

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <sys/time.h>
#include <pthread.h>

int put_select_single_np(
        int nfds,
        fd_set *readfds,
        fd_set *writefds,
        fd_set *errorfds,
        struct timeval *timeout);
```

## PARAMETERS

See the **select(2)** reference page.

## DESCRIPTION

The **put_select_single_np( )** function is a thread-aware version of the **select( )** function used to check the status of a single file descriptor.

To improve application performance, use the **put_select_single_np( )** function instead of the default thread-aware **select( )** function that is mapped by the _PUT_MODEL_ feature test macro. For multiple file desciptors, use the default thread-aware **select( )** function mapped by the _PUT_MODEL_ feature test macro.

In **sys/time.h**, a mapping of **select( )** to **put_select_single_np( )** has been defined:

**#pragma function select (alias("put_select_single_np"), unspecified)**

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## NOTES

To use a combination of the default thread-aware **select( )** and the **put_select_single_np( )** functions in a single source file, you must compile the application using the **_PUT_MODEL_** feature test macro only and explicitly call **put_select_single_np( )**.

## RETURN VALUES

See the **select(2)** reference page. The following information also applies:

- If the file descriptor becomes invalid (is closed by another thread), -1 is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## RELATED INFORMATION

Functions: **select(2)**.

## NAME

**PUT_SETMODE -** Sets device-dependent Guardian file-system functions

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_ SETMODE(
        short filenum,
        short function,
        short param1,
        short param2,
        short *last_params);
```

## PARAMETERS

*filenum* — Specifies the Guardian file number of a Guardian file open instance, identifying the file to receive the requested function.

*function* — Specifies the number of a device-dependent function. For a description of valid values, see the table of SETMODE functions in the *Guardian Procedure Calls Reference Manual*.

*param1* — (Optional) Provides the first value or pattern of set bits that defines the specific function setting to be used. For a description of valid values, see the table of SETMODE functions in the *Guardian Procedure Calls Reference Manual*.

*param2* — (Optional) Provides the second value or pattern of set bits that defines the specific function setting to be used. For a description of valid values, see the table of SETMODE functions in the *Guardian Procedure Calls Reference Manual*.

*last_params* — (Optional) Returns the previous settings of *param1* and *param2* associated with the current function.

## DESCRIPTION

The **PUT_SETMODE( )** function is a thread-aware version of the Guardian SETMODE procedure.

The **PUT_SETMODE( )** function is used to set device-dependent Guardian file-system functions. A call to the **PUT_SETMODE( )** function is rejected with an error indication if incomplete nowait operations are pending on the specified file.

For programming information about the Guardian SETMODE file-system procedure, see the *Guardian Programmer's Guide* and the manual for the data communication protocol you are using.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## Considerations

Default settings

The **PUT_SETMODE( )** settings designated as default in the *Guardian Procedure Calls Reference Manual* are the values that apply when a file is opened (not if a particular *function* value is omitted when **PUT_SETMODE( )** is called).

Waited **PUT_SETMODE( )** use

The **PUT_SETMODE( )** function is used on a file as a waited operation even if *filenum* has been opened for nowait operations. Use the Guardian SETMODENOWAIT procedure for nowait operations.

Use for Telserv processes

No **PUT_SETMODE( )** calls on Telserv are allowed before doing an **PUT_CONTROL( )** function 11.

Ownership and security of a disk file

"Set disk file security" and "set disk file owner" are rejected unless the requester is the owner of the file or the super ID.

## Interprocess Communication Considerations

Nonstandard parameter values

You can specify any value for the *function*, *param1*, and *param2* parameters. Establish an application-defined protocol for interpreting nonstandard parameter values.

User-defined functions

Use of *function* code numbers 100 to 109 avoids any potential conflict with **PUT_SETMODE( )** function codes defined by HP.

Incorrect use of *last_params*

Guardian file-system error 2 is returned when the *last_params* parameter is supplied but the target process does not correctly return values for this parameter.

Process message

Issuing a **PUT_SETMODE( )** call to a file representing another process causes a system message -33 (process SETMODE) to be sent to that process.

You can identify the process that called **PUT_SETMODE( )** in a subsequent call to the Guardian FILE_GETRECEIVEINFO_ (or LASTRECEIVE or RECEIVEINFO) procedure. For a list of all system messages sent to processes, see the *Guardian Procedure Errors and Messages Manual*.

**RETURN VALUES**

The **PUT_SETMODE( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

**ERRORS**

None. This function does not set the **errno** variable.

**RELATED INFORMATION**

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**put_setOSSFileIOHandler -** Sets interest in file descriptor

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**extern put_error_t put_setOSSFileIOHandler(**
**const int** *filedes***,**
**const int** *read***,**
**const int** *write***,**
**const int** *error***);**

## PARAMETERS

*filedes* Specifies the OSS file descriptor for the file of interest.

*read* Nonzero indicates interest in read ready.

*write* Nonzero indicates interest in write ready.

*error* Nonzero indicates interest in exception pending.

## DESCRIPTION

This function sets interest in an OSS file descriptor.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

**PUT_SUCCESS**

This value is returned for any of the following conditions:

- The *filedes* interest was successfully set
- The *filedes* was not registered prior to this call
- The specified *filedes* is invalid
- The specified *filedes* is not supported

**PUT_ERROR** The specified *filedes* was less than 0 (zero).

## NAME

**put_setTMFConcurrentTransactions -** Sets the number of concurrent TMF transactions

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <pthread.h>

int put_setTMFConcurrentTransactions(
        short max_trans);
```

## PARAMETERS

*max_trans* Specifies the maximum number of concurrent transactions desired.

## DESCRIPTION

This function sets the maximum number of concurrent TMF transactions.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

This function returns 0 (zero) upon successful completion of the call. If an error occurs, this function can return the following value:

**EINVAL** Unable to change the maximum number of concurrent transactions because TMF is already processing transactions.

## RELATED INFORMATION

Functions: **put_getTMFConcurrentTransactions(2)**.

## NAME

**PUT_TMF_GetTxHandle -** Gets the current TMF transaction handle

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <pthread.h>

short PUT_TMF_GetTxHandle(
        PUT_TMF_TxHandle_t *tx_handle);
```

## PARAMETERS

*tx_handle* Receives the current active TMF transaction handle.

## DESCRIPTION

This function retrieves the current active transaction handle of the thread.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

This function returns an integer value indicating the result of the call. Possible return values are:

**0** (zero) Successful completion of the call. The current active transaction handle is returned in *tx_handle*.

**22** A bounds error occurred.

**29** There are missing parameters.

**75** There is no current transaction.

**RELATED INFORMATION**

Functions: **PUT_TMF_SetTxHandle(2)**, **PUT_TMF_Init(2)**.

## NAME

**PUT_TMF_Init -** Initializes the tfile for concurrent transaction management

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**short PUT_TMF_Init(void);**

## PARAMETERS

None.

## DESCRIPTION

This function opens the tfile for concurrent transaction management.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

**PUT_SUCCESS**
The TMF file is initialized for concurrent transaction management.

*error* Contains the error value returned by the underlying call to the Guardian OPEN procedure. See the *Guardian Procedure Errors and Messages Manual* for more information on the specific value returned.

## RELATED INFORMATION

Functions: **PUT_TMF_GetTxHandle(2)**, **PUT_TMF_SetTxHandle(2)**, **put_getTMFConcurrentTransactions(2)**, **put_setTMFConcurrentTransactions(2)**.

## NAME

**PUT_TMF_RESUME -** Resumes a previously suspended transaction associated with the current thread

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**short PUT_TMF_RESUME(**
    **long long** **txid***);**

## PARAMETERS

### Input

*txid* Specifies the transactional identifier returned by **PUT_TMF_SUSPEND( )** or TMF_GET_TX_ID.

## DESCRIPTION

This function resumes a previously suspended transaction associated with the current thread.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

A status word is returned. The value is one of the following:

0 (zero) The **PUT_TMF_RESUME( )** operation completed successfully.

Nonzero values
The Guardian file-system error with this error number occurred.

**RELATED INFORMATION**

Functions: **PUT_TMF_SUSPEND(2)**.

## NAME

**PUT_TMF_SetAndValidateTxHandle -** Sets the current TMF transaction handle to be associated with the current thread

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**short PUT_TMF_SetAndValidateTxHandle(**
    **PUT_TMF_TxHandle_t ****tx_handle***);**

## PARAMETERS

*tx_handle* Specifies the transaction handle of the current TMF transaction.

## DESCRIPTION

This function sets the specified transaction handle as the current active transaction for the thread. In addition, it validates the transaction. If the transaction is not valid, the transaction is aborted.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

This function returns an integer value indicating the result of the call. Possible return values are:

0 (zero) The **PUT_TMF_SetAndValidateTxHandle( )** operation completed successfully; the transaction handle was successfully set and validated.

Nonzero values
The Guardian file-system error with this error number occurred.

**RELATED INFORMATION**

Functions: **PUT_TMF_GetTxHandle(2)**, **PUT_TMF_SetTxHandle(2)**, **PUT_TMF_Init(2)**.

## NAME

**PUT_TMF_SetTxHandle -** Sets the TMF transaction handle

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**short PUT_TMF_SetTxHandle(**
**PUT_TMF_TxHandle_t ****tx_handle***);**

## PARAMETERS

*tx_handle* Specifies the transaction handle of the current TMF transaction.

## DESCRIPTION

This function sets the specified transaction handle as the current active transaction for the thread.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

This function returns an integer value indicating the result of the call. Possible return values are:

**0** (zero) Indicates the transaction handle was successfully set.

**22** Indicates that a bounds error occurred.

**29** Indicates missing parameters.

**75** Indicates that there is no current transaction.

**78** Indicates an invalid transaction identifier or that a transaction has not started on this Expand node.

**715** Indicates an invalid transaction handle.

**RELATED INFORMATION**

Functions: **PUT_TMF_GetTxHandle(2)**, **PUT_TMF_Init(2)**.

## NAME

**PUT_TMF_SUSPEND -** Suspends a transaction associated with the current thread

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**short PUT_TMF_SUSPEND(**
**long long** **txid***);**

## PARAMETERS

### Output

*txid* Returns a transactional identifier that can be used for a subsequent **PUT_TMF_RESUME( )** call.

## DESCRIPTION

This function suspends a transaction associated with the current thread.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

A status word is returned. The value is one of the following:

0 (zero) The **PUT_TMF_SUSPEND( )** operation completed successfully.

Nonzero values
The Guardian file-system error with this error number occurred.

## RELATED INFORMATION

Functions: **PUT_TMF_RESUME(2)**.

## NAME

**PUT_UNLOCKFILE -** Unlocks a disk file and any records in that file currently locked by the user

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <pthread.h>**

**short PUT_UNLOCKFILE(**
    **short** *filenum***,**
    **long** *tag***);**

## PARAMETERS

*filenum* — Specifies the Guardian file number of a Guardian file open instance for the file that you want unlocked.

*tag* — (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **PUT_UNLOCKFILE( )** function is a thread-aware version of the Guardian UNLOCKFILE procedure.

The **PUT_UNLOCKFILE( )** function unlocks a disk file and any records in that file currently locked by the user. The user is defined either as the opener of the file (identified by the *filenum* value used) if the file is not audited, or by the transaction (identified by the TRANSID) if the file is audited. Unlocking a file allows other processes to access the file. This call has no affect on an audited file if the current transaction has modified that file.

For programming information about the Guardian UNLOCKFILE file-system procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

### Considerations

Nowait and **PUT_UNLOCKFILE()**
: The **PUT_UNLOCKFILE()** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

Locking queue
: If any users are queued in the locking queue for the file, the process at the head of the locking queue is granted access and is removed from the queue (the next read or lock request moves to the head of the queue). If the next user in the locking queue is waiting to:

  - lock the file or lock a record in the file, the user is granted the lock (which excludes other users from accessing the file) and resumes processing.
  - read the file, its read is processed.

Transaction Management Facility (TMF) and **PUT_UNLOCKFILE()**
: If the current transaction modifies a file audited by TMF, locks on the file are released only when TMF ends or aborts the transaction. In other words, a locked audited file that the current transaction modified is unlocked during **PUT_ENDTRANSACTION()** or **PUT_ABORTTRANSACTION()** processing for that file. You can use the **PUT_UNLOCKFILE()** function to unlock an unmodified audited record.

### Use on OSS Objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **PUT_UNLOCKFILE()** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**PUT_UNLOCKREC -** Unlocks a Guardian file record currently locked by the user

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <pthread.h>**

**short PUT_UNLOCKREC(**
**short** *filenum***,**
**long** *tag***);**

## PARAMETERS

*filenum* Specifies the Guardian file number of a Guardian file open instance for the file containing the record you want unlocked.

*tag* (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **PUT_UNLOCKREC( )** function is a thread-aware version of the Guardian UNLOCKREC procedure.

The **PUT_UNLOCKREC( )** function unlocks a record in the specified file currently locked by the user. The user is defined either as the opener of the file (identified by the *filenum* value used) if the file is not audited, or by the transaction (identified by the TRANSID) if the file is audited.

This call unlocks the record at the current position in the file, allowing other users to access that record. This call has no affect on a record of an audited file if the current transaction has modified that record.

For programming information about the Guardian UNLOCKREC file-system procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

### Considerations

File opened nowait and **PUT_UNLOCKREC( )**
: The **PUT_UNLOCKREC( )** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

Locking queue
: If any users are queued in the locking queue for the record, the user at the head of the locking queue is granted access and is removed from the queue (the next read or lock request moves to the head of the queue).

: If the user granted access is waiting to lock the record, the user is granted the lock (which excludes other process from accessing the record) and resumes processing. If the user granted access is waiting to read the record, its read is processed.

Calling **PUT_UNLOCKREC( )** after KEYPOSITION
: If the call to **PUT_UNLOCKREC( )** immediately follows a call to KEYPOSITION where a nonunique alternate key is specified, the **PUT_UNLOCKREC( )** call fails. A subsequent call to FILE_GETINFO_ or FILEINFO shows that Guardian file-system error 46 (`invalid key`) occurred. However, if an intermediate call to **PUT_READX( )** or **PUT_READLOCKX( )** is performed, the call to **PUT_UNLOCKREC( )** is permitted.

Unlocking several records
: If several records need to be unlocked, you can call the **PUT_UNLOCKREC( )** function to unlock all records currently locked by the user (rather than unlocking the records through individual calls to **PUT_UNLOCKREC( )**).

Current-state indicators after **PUT_UNLOCKREC( )**
: For key-sequenced, relative, and entry-sequenced files, the current-state indicators after an UNLOCKREC remain unchanged.

File pointers after **PUT_UNLOCKREC( )**
: For unstructured files, the current-record pointer and the next-record pointer remain unchanged.

Transaction Management Facility (TMF) and **PUT_UNLOCKREC( )**
: If the current transaction modifies a record in file audited by TMF, locks on the record are released only when TMF ends or aborts the transaction. In other words, a locked record in an audited file that the current transaction modified is unlocked during **PUT_ENDTRANSACTION( )** or **PUT_ABORTTRANSACTION( )** processing for that file. You can use the **PUT_UNLOCKREC( )** function to unlock an unmodified audited record.

### Use on OSS Objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **PUT_UNLOCKREC( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**put_unregFile -** Unregisters a Guardian file number as one that the user manages

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <pthread.h>

extern put_error_t put_unregFile(
        const short filenum);
```

## PARAMETERS

*filenum* Specifies the Guardian file number being unregistered.

## DESCRIPTION

This function unregisters a Guardian file number as one that the user manages. Any threads waiting on file number I/O will awaken with **PUT_ERROR** and Guardian file-system error 16.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

**PUT_SUCCESS**
The specified *filenum* was successfully unregistered.

**PUT_ERROR** One of the following conditions exists:

- The value specified for *filenum* s less than 0 (zero).
- The specified *filenum* was not registered prior to this call.

- The FILE_COMPLETE_SET_ procedure removal of *filenum* returned a nonzero value.

## NAME

**put_unregOSSFileIOHandler -** Unregisters an OSS file descriptor

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**extern put_error_t put_unregOSSFileIOHandler(**
**const int** *filedes***);**

## PARAMETERS

*filedes* Specifies the OSS file descriptor being unregistered.

## DESCRIPTION

This function unregisters an OSS file descriptor as one that the user manages.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

**PUT_SUCCESS**
The specified *filedes* was successfully unregistered.

**PUT_ERROR** The specified *filedes* is less than 0 (zero) or was not registered prior to this call.

## NAME

**put_unregPathsendTagHandler -** Unregisters the user-supplied Pathsend tag

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**put_error_t put_unregPathsendTagHandler(**
**const long** *tag***);**

## PARAMETERS

*tag* Specifies the Pathsend tag to be unregistered.

## DESCRIPTION

This function unregisters the specified Pathsend tag as a tag that user manages.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## RETURN VALUES

**PUT_SUCCESS**
The specified tag was unregistered.

**PUT_ERROR** The specified tag was never registered.

## RELATED INFORMATION

Functions: **put_regPathsendTagHandler(2)**, **PUT_SERVERCLASS_DIALOG_ABORT_(2)**, **PUT_SERVERCLASS_DIALOG_BEGIN_(2)**, **PUT_SERVERCLASS_DIALOG_END_(2)**, **PUT_SERVERCLASS_DIALOG_SEND_(2)**, **PUT_SERVERCLASS_SEND_INFO_(2)**, **PUT_SERVERCLASS_SEND_(2)**.

## NAME

**put_wakeup** - Wakes up a thread awaiting tagged I/O

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <pthread.h>**

**extern put_error_t put_wakeup(**
 **const short** *filenum***,**
 **const long** *tag***,**
 **const long** *count_transferred***,**
 **const long** *error***);**

## PARAMETERS

*filenum* Specifies the Guardian file number being waited on.

*tag* Specifies the tag that is being awaited; the value -1 indicates all tags.

*count_transferred*
Specifies byte transfer count of completed I/O.

*error* Specifies Guardian error number for IO.

## DESCRIPTION

This function wakes up a thread awaiting the tagged I/O on the file with the specified Guardian file number. The awakened thread returns from its call to the **put_awaitio( )** function with a return value of **PUT_SUCCESS**.

To use this function on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

**RETURN VALUES**

**PUT_SUCCESS**

One of the following conditions exists:

- *tag* was not -1 and waiting I/O was awakened. Note that only one awaiting I/O was awakened.
- *tag* was -1 and awaiting I/O (if any) was awakened.

**PUT_ERROR** One of the following conditions exists:

- The value specified for *filenum* was less than 0 (zero).
- *tag* was not -1 and no awaiting IO was found.

## NAME

**PUT_WRITEREADX -** Writes data to a Guardian file from an array and waits for data to be read back from the file

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_WRITEREADX(
        short filenum,
        char *buffer,
       #ifdef __LP64
        int write_count,
        int read_count,
        int *count_read,
       #else
        unsigned short write_count,
        unsigned short read_count,
        unsigned short *count_read,
       #endif
        long tag);
```

## PARAMETERS

### Input

*filenum* Specifies the file number of a Guardian file open instance that identifies the file to be read.

*write_count* Specifies the number of bytes to be written.

*read_count* Specifies the number of bytes to be read.

*tag* (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

### Output

*buffer* Specifies an array in the application process in which the information to be written to the file is stored before the call. On return, *buffer* contains the information read from the file.

*count_read* (Optional) For waited I/O only. This parameter returns a count of the number of bytes returned from the file into *buffer*.

## DESCRIPTION

The **PUT_WRITEREADX( )** function is a thread-aware version of the Guardian WRITEREADX procedure.

The **PUT_WRITEREADX( )** function writes data to a file from an array in the application process, then waits for data to be transferred back from the file. The data from the read portion returns in the same array used for the write portion.

If the file is opened for nowait I/O, you must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This condition also applies to other processes that might be sharing the segment. The application must ensure that the buffer used in the call to the **PUT_WRITEREADX( )** function is not reused before the I/O completes with a call to AWAITIOX.

For programming information about the WRITEREADX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## Considerations

Buffer use
: **PUT_WRITEREADX( )** is intended for use with 32-bit extended addresses and 64-bit extended addresses. The data buffer for **PUT_WRITEREADX( )** can be either in the caller's stack segment or any extended data segment.

Terminals
: A special hardware feature is incorporated in the asynchronous multiplexer controller that ensures the system is ready to read from the terminal as soon as the write is completed.

Interprocess communication
: The **PUT_WRITEREADX( )** function is used to originate a message to another process that was previously opened, then waits for a reply from that process.

Waited I/O read operation
: If a waited I/O **PUT_WRITEREADX( )** call is executed, the *count_read* parameter indicates the number of bytes actually read.

Nowait I/O read operation
: If a nowait I/O **PUT_WRITEREADX( )** call is executed, *count_read* has no meaning and can be omitted. The count of the number of bytes read is obtained when the I/O operation completes through the *count-transferred* parameter of the Guardian AWAITIOX procedure.

: The **PUT_WRITEREADX( )** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

Do not change the contents of the data buffer between the initiation and completion of a nowait **PUT_WRITEREADX( )** operation. A retry can copy the data again from the user buffer and cause the wrong data to be written. Avoid sharing a buffer between a **PUT_WRITEREADX( )** and another I/O operation because the contents of the data buffer might change before the write is completed.

Carriage return/line feed sequence after the write

No carriage return and line feed sequence is sent to the terminal after the write part of the operation.

Location of *buffer* and *count_read*

The buffer and count transferred can be in the user stack or in an extended data segment. The *buffer* and *count_read* cannot be in the user code space.

If the *buffer* and *count_read* are in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **PUT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This restriction also applies to other processes that might share the segment. It is the application's responsibility to ensure this.
- If you initiated the I/O with **PUT_WRITEREADX( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- A selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- You can call **PUT_CANCEL( )** or CANCELREQ to cancel nowait I/O initiated with **PUT_WRITEREADX( )**. The I/O is canceled if the file is closed before the I/O completes or if you call the Guardian AWAITIOX procedure with a positive time limit and specific file number and the request times out.

Bounds checking

If the extended address of *buffer* is odd, bounds checking rounds the address to the next lower word boundary and also checks an extra byte. The odd address is used for the transfer.

## RETURN VALUES

The **PUT_WRITEREADX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**PUT_WRITEUPDATEUNLOCKX -** Performs random processing of records in a disk file

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_WRITEUPDATEUNLOCKX(
     short filenum,
     char *buffer,
    #ifdef __LP64
     int write_count,
     int *count_written,
    #else
     unsigned short write_count,
     unsigned short *count_written,
    #endif
     long tag);
```

## PARAMETERS

### Input

*filenum* Specifies the file number of a Guardian file open instance that identifies the file to be written.

*buffer* Specifies an array in the application process in which the information to be written to the file is stored before the call.

*write_count* Specifies the number of bytes to be written.

*tag* (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

### Output

*count_written* (Optional) Returns a count of the number of bytes written to the file from *buffer*.

## DESCRIPTION

The **PUT_WRITEUPDATEUNLOCKX( )** function is a thread-aware version of the Guardian WRITEUPDATEUNLOCKX procedure.

The **PUT_WRITEUPDATEUNLOCKX( )** function performs random processing of records in a Guardian disk file. **PUT_WRITEUPDATEUNLOCKX( )** has two purposes:

- To alter, then unlock, the contents of the record at the current position
- To delete the record at the current position in a key-sequenced or relative file

A call to **PUT_WRITEUPDATEUNLOCKX( )** is equivalent to a call to **PUT_WRITEUPDATEX( )** followed by a call to **PUT_UNLOCKREC( )**. However, the **PUT_WRITEUPDATEUNLOCKX( )** function requires less system processing than do the separate calls to **PUT_WRITEUPDATEX( )** and **PUT_UNLOCKREC( )**.

For programming information about the WRITEUPDATEUNLOCKX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## Considerations

Buffer use

**PUT_WRITEUPDATEUNLOCKX( )** is intended for use with 32-bit extended addresses and 64-bit extended addresses. The data buffer for **PUT_WRITEUPDATEUNLOCKX( )** can be either in the caller's stack segment or any extended data segment.

Nowait I/O and **PUT_WRITEUPDATEUNLOCKX( )**

The **PUT_WRITEUPDATEUNLOCKX( )** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

For files audited by the Transaction Management Facility (TMF), You must call the AWAITIOX procedure to complete the **PUT_WRITEUPDATEUNLOCKX( )** operation before **PUT_ENDTRANSACTION( )** or **PUT_ABORTTRANSACTION( )** is called.

Do not change the contents of the data buffer between the initiation and completion of a nowait write operation. A retry can copy the data again from the user buffer and cause the wrong data to be written. Avoid sharing a buffer between a write and another I/O operation because this creates the contents of the write buffer might change before the write is completed.

Random processing and **PUT_WRITEUPDATEUNLOCKX( )**

For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. Positioning for **PUT_WRITEUPDATEUNLOCKX( )** is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to **PUT_WRITEUPDATEUNLOCKX( )** is rejected with

Guardian file-system error 11 (`record does not exist`).

Unstructured files (pointers unchanged)
For unstructured files, data is written in the position indicated by the current-record pointer. A call to **PUT_WRITEUPDATEUNLOCKX( )** for an unstructured file typically follows a call to the Guardian POSITION procedure or **PUT_READUPDATEX( )**. The current-record and next-record pointers are not changed by a call to **PUT_WRITEUPDATEUNLOCKX( )**.

How **PUT_WRITEUPDATEUNLOCKX( )** works
The record unlocking performed by **PUT_WRITEUPDATEUNLOCKX( )** functions in the same manner as **PUT_UNLOCKREC( )**.

`Record does not exist`
Positioning for **PUT_WRITEUPDATEUNLOCKX( )** is always to the record described by the exact value of the current key and current-key specifier. Therefore, if such a record does not exist, the call to **PUT_WRITEUPDATEUNLOCKX( )** is rejected with Guardian file-system error 11.

Invalid write operations to queue files
DP2 rejects **PUT_WRITEUPDATEUNLOCKX( )** operations with a Guardian file-system error 2.

Location of *buffer* and *count_written*
The buffer and count transferred can be in the user stack or in an extended data segment. The *buffer* and *count_written* cannot be in the user code space.

If the *buffer* and *count_written* are in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **PUT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This restriction also applies to other processes that might be sharing the segment. It is the application's responsibility to ensure this.
- If you initiated the I/O with **PUT_WRITEUPDATEUNLOCKX( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- A selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- Nowait I/O initiated with **PUT_WRITEUPDATEUNLOCKX( )** can be canceled with a call to **PUT_CANCEL( )** or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or if the Guardian AWAITIOX procedure is called with a positive time limit and specific file number and the request times out.

Bounds checking

If the extended address of *buffer* is odd, bounds checking rounds the address to the next lower word boundary and also checks an extra byte. The odd address is used for the transfer.

All considerations for **PUT_WRITEUPDATEX( )** also apply to this call.

### Use on OSS Objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **PUT_WRITEUPDATEUNLOCKX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEX(2)**, **PUT_WRITEX(2)**.

## NAME

**PUT_WRITEUPDATEX -** Transfers data from an array in the application program to a Guardian file

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_WRITEUPDATEX(
        short filenum,
        char *buffer,
       #ifdef __LP64
        int write_count,
        int *count_written,
       #else
        unsigned short write_count,
        unsigned short *count_written,
       #endif
        long tag);
```

## PARAMETERS

### Input

*filenum* Specifies the file number of a Guardian file open instance that identifies the file to be written.

*buffer* Specifies an array in the application process in which the information to be written to the file is stored before the call.

*write_count* Specifies the number of bytes to be written.

*tag* (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

### Output

*count_written* (Optional) Returns a count of the number of bytes written to the file from *buffer*.

## DESCRIPTION

The **PUT_WRITEUPDATEX( )** function is a thread-aware version of the Guardian WRITEUPDATEX procedure.

The **PUT_WRITEUPDATEX( )** function performs random processing of records in a Guardian disk file. **PUT_WRITEUPDATEX( )** has two purposes:

- To alter the contents of the record at the current position
- To delete the record at the current position in a key-sequenced or relative file

Data from the application process's array is written in the position indicated by the setting of the current-record pointer. A call to this procedure typically follows a corresponding call to the **PUT_READX( )** or **PUT_READUPDATEX( )** function. The current-record and next-record pointers are not affected by the **PUT_WRITEUPDATEX( )** procedure.

For magnetic tapes, **PUT_WRITEUPDATEX( )** is used to replace a record in an already written tape. The tape is backspaced one record; the data from the application process's array is written in that area.

For programming information about the WRITEUPDATEX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## Considerations

Buffer use

**PUT_WRITEUPDATEX( )** is intended for use with 32-bit extended addresses and 64-bit extended addresses. The data buffer for **PUT_WRITEUPDATEX( )** can be either in the caller's stack segment or any extended data segment.

I/O counts with unstructured files

Unstructured files are transparently blocked using one of the four valid block sizes (512, 1024, 2048, or 4096 bytes; 4096 is the default). This transparent block size, known as BUFFERSIZE, is the transfer size used against an unstructured file. While BUFFERSIZE does not change the maximum unstructured transfer (4096 bytes), multiple I/O operations might be performed to satisfy a user's request depending on the BUFFERSIZE chosen. For example, if BUFFERSIZE is 512 bytes, and a request is made to write 4096 bytes, at least eight transfers, each 512 bytes long, will be made. More than eight transfers happen, in this case, if the requested transfer does not start on a BUFFERSIZE boundary.

DP2 performance with unstructured files is best when requested transfers begin on BUFFERSIZE boundaries and are integral multiples of BUFFERSIZE.

Because the maximum blocksize for DP2 structured files is also 4096 bytes, this is also the maximum structured transfer size for DP2.

Deleting locked records

Deleting a locked record implicitly unlocks that record unless the file is audited, in which case the lock is not removed until the transaction terminates.

Waited **PUT_WRITEUPDATEX( )** calls

If a waited **PUT_WRITEUPDATEX( )** call is executed, the *count_written* parameter indicates the number of bytes actually written.

Nowait **PUT_WRITEUPDATEX( )** calls

If a nowait **PUT_WRITEUPDATEX( )** call is executed, *count_written* has no meaning and can be omitted. The count of the number of bytes written is obtained through the *count-transferred* parameter of the Guardian AWAITIOX procedure when the I/O completes.

The **PUT_WRITEUPDATEX( )** procedure must finish with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O. For files audited by the Transaction Management Facility (TMF), the AWAITIOX procedure must be called before the **PUT_ENDTRANSACTION( )** or **PUT_ABORTTRANSACTION( )** function is called.

Do not change the contents of the data buffer between the initiation and completion of a nowait write operation. A retry can copy the data again from the user buffer and cause the wrong data to be written. Avoid sharing a buffer between a write and another I/O operation because the contents of the write buffer might change before the write is completed.

Invalid write operations to queue files

Attempts to perform **PUT_WRITEUPDATEX( )** operations are rejected with a Guardian file-system error 2.

**Disk File Considerations**

Large data transfers

To enable large data transfers (more than 4096 bytes), you can use **PUT_SETMODE( )** function 141. See the description of SETMODE functions in the *Guardian Procedure Calls Reference Manual*.

Random processing and **PUT_WRITEUPDATEX( )**

For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. Positioning for **PUT_WRITEUPDATEX( )** is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to **PUT_WRITEUPDATEX( )** is rejected with Guardian file-system error 11 (`record does not exist`).

`File is locked`

If a call to **PUT_WRITEUPDATEX( )** is made and the file is locked through a file number other than that supplied in the call, the call is rejected with Guardian file-system error 73 (`file is locked`).

When the just-read record is updated

A call to **PUT_WRITEUPDATEX( )** following a call to **PUT_READX( )**, without intermediate positioning, updates the record just read.

Unstructured files

Transferring disk file data

If the **PUT_WRITEUPDATEX( )** call is to an unstructured disk file, data is transferred to the record location specified by the current-record pointer.

File pointers after a successful call

After a successful **PUT_WRITEUPDATEX( )** call to an unstructured file, the current-record and next-record pointers are unchanged.

Number of bytes written

If the unstructured file is created with the odd unstructured attribute (also known as ODDUNSTR) set, the number of bytes written is exactly the number specified in *write_count*. If the odd unstructured attribute is not set when the file is created, the value of *write_count* is rounded up to an even value before the **PUT_WRITEUPDATEX( )** call is executed.

You set the odd unstructured attribute with the Guardian FILE_CREATE_, FILE_CREATELIST_, or CREATE procedure, or with the File Utility Program (FUP) SET and CREATE commands.

Structured files

Calling **PUT_WRITEUPDATEX( )** after KEYPOSITION

If the call to **PUT_WRITEUPDATEX( )** immediately follows a call to the Guardian KEYPOSITION procedure in which a nonunique alternate key is specified as the access path, the **PUT_WRITEUPDATEX( )** call fails. A subsequent call to the Guardian FILE_GETINFO_ or FILEINFO procedure shows that Guardian file-system error 46 (`invalid key`) occurred. However, if an intermediate call to **PUT_READX( )** or **PUT_READLOCKX( )** is performed, the call to **PUT_WRITEUPDATEX( )** is permitted because a unique record is identified.

Specifying *write_count* for entry-sequenced files

For entry-sequenced files, the value of *write_count* must match exactly the *write_count* value specified when the record was originally inserted into the file.

Changing the primary-key of a key-sequenced record

An update to a record in a key-sequenced file cannot alter the value of the primary-key field. To change the primary-key field, you must delete the old record (**PUT_WRITEUPDATEX( )** with *write_count* = 0 [zero]) and insert a new record with the key field changed (**PUT_WRITEX( )**).

Current-state indicators after **PUT_WRITEUPDATEX( )**

After a successful **PUT_WRITEUPDATEX( )** call, the current-state indicators remain unchanged.

The buffer and count transferred can be in the user stack or in an extended data segment. The buffer and count transferred cannot be in the user code space.

If the buffer or count transferred is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **PUT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to AWAITIOX. This also applies to other processes that might be sharing the segment. It is the application's responsibility to ensure this.
- If you initiated the I/O with **PUT_WRITEUPDATEX( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- The extended segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- Nowait I/O initiated with **PUT_WRITEUPDATEX( )** can be canceled with a call to the **PUT_CANCEL( )** function or the Guardian CANCELREQ procedure. The I/O is canceled if the file is closed before the I/O completes or AWAITIOX is called with a positive time limit and specific file number and the request times out.

Bounds checking

If the extended address of the buffer is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

**Magnetic Tape Considerations**

Supported equipment

**PUT_WRITEUPDATEX( )** is permitted only on the 3202 Controller for the 5103 or 5104 Tape Drives. This function is not supported on any other controller/tape drive combination. **PUT_WRITEUPDATEX( )** is specifically not permitted on the following controller/tape drive pairs:

- 3206 Controller and the 5106 Tri-Density Tape Drive
- 3207 Controller and the 5103 & 5104 Tape Drives
- 3208 Controller and the 5130 & 5131 Tape Drives

Specifying the correct number of bytes written

When **PUT_WRITEUPDATEX( )** is used with magnetic tape, the number of bytes to be written must fit exactly; otherwise, information on the tape can be lost. However, no error indication is given.

Limitation of **PUT_WRITEUPDATEX( )** to the same record
Five is the maximum number of times a **PUT_WRITEUPDATEX( )** call can be executed to the same record on tape.

**RETURN VALUES**

The **PUT_WRITEUPDATEX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

**ERRORS**

None. This function does not set the **errno** variable.

**RELATED INFORMATION**

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEX(2)**.

## NAME

**PUT_WRITEX -** Writes data from an array in the application program to an open Guardian file

## LIBRARY

32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <pthread.h>

short PUT_WRITEX(
        short filenum,
        char *buffer,
       #ifdef __LP64
        int write_count,
        int *count_written,
       #else
        unsigned short write_count,
        unsigned short *count_written,
       #endif
        long tag);
```

## PARAMETERS

### Input

*filenum* Specifies the file number of a Guardian file open instance that identifies the file to be written.

*buffer* Specifies an array in the application process in which the information to be written to the file is stored before the call.

*write_count* Specifies the number of bytes to be written.

*tag* (Optional) For nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

### Output

*count_written* (Optional) Returns a count of the number of bytes written to the file from *buffer*.

## DESCRIPTION

The **PUT_WRITEX( )** function is a thread-aware version of the Guardian WRITEX procedure.

This function writes data from an array in the application program to an open Guardian file.

For programming information about the WRITEX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

To use this function on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Link the application to the **zputdll** library.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit applications.

To use this function in a 32-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

## Considerations

Buffer use
: **PUT_WRITEX( )** is intended for use with 32-bit extended addresses and 64-bit extended addresses. The data buffer for **PUT_WRITE( )** can be either in the caller's stack segment or any extended data segment.

Waited I/O and **PUT_WRITEX( )** calls
: If a waited **PUT_WRITEX( )** call is executed, the *count_written* parameter indicates the number of bytes actually written.

Nowait I/O and **PUT_WRITEX( )** calls
: If a nowait **PUT_WRITE ( )** call is executed, *count_written* has no meaning and can be omitted. The count of the number of bytes written is obtained when the I/O operation completes through the *count-transferred* parameter of the Guardian AWAITIOX procedure.

: The **PUT_WRITEX( )** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

: Do not change the contents of the data buffer between the initiation and completion of a nowait write operation. A retry can copy the data again from the user buffer and cause the wrong data to be written. Avoid sharing a buffer between a write and another I/O operation because the contents of the write buffer might change before the write is completed.

## Disk File Considerations

Large data transfers for unstructured files using default mode
: Default mode allows I/O sizes for unstructured files to be as large as 56KB (57,344), excepting writes to audited files, if the unstructured buffer size (or block size) is 4KB (4096). Default mode refers to the mode of the file if **PUT_SETMODE( )** function 141 is not invoked.

: For an unstructured file with an unstructured buffer size other than 4KB, DP2 automatically adjusts the unstructured buffer size to 4KB, if possible, when an I/O larger than 4KB is attempted. However, this adjustment is not possible for files that have extents with an odd number of pages; in such cases, an I/O over 4KB is not possible. The switch to a different unstructured buffer size will have a transient performance impact, so HP recommends that the size be initially set to 4KB, which is the default. Transfer sizes over 4KB are not supported in default mode for unstructured access to structured files.

Large data transfers using **PUT_SETMODE(141)**
: You can use **PUT_SETMODE()** function 141 to enable large data transfers (more than 4096 bytes) for files opened with unstructured access, regardless of unstructured buffer size. When you use **PUT_SETMODE(141)** to enable large data transfers, you can to specify up to 56K (57344) bytes for the *write_count* parameter. See the description of SETMODE functions in the *Guardian Procedure Calls Reference Manual*.

`File is locked`
: If you call **PUT_WRITEX()** is made and the file is locked through a file number other than that supplied in the call, the call is rejected with Guardian file-system error 73 (`file is locked`).

Inserting a new record into a file
: The **PUT_WRITEX()** function inserts a new record into a file in the position designated by the file's primary key:

Key-sequenced files
: The record is inserted in the position indicated by the value in its primary-key field.

Queue files
: The record is inserted into a file at a unique location. The disk process sets the timestamp field in the key, which causes the record to be positioned after the other existing records that have the same high-order user key.

If the file is audited, the record is available for read operations when the transaction associated with the write operation commits. If the transaction aborts, the record is never available to read operations. If the file is not audited, the record is available as soon as the write operation finishes successfully. Unlike other key-sequenced files, a write operation to a queue file will never encounter a Guardian file-system error 10 (`duplicate record`) because all queue file records have unique keys generated for them.

Relative files
: After an open or an explicit positioning by its primary key, the record is inserted in the designated position.

Subsequent **PUT_WRITEX()** calls without intermediate positioning insert records in successive record positions. If -2 is specified in a preceding positioning, the record is inserted in an available record position in the file.

If -1 is specified in a preceding positioning, the record is inserted following the last position used in the file. An existing record does not have to be in that position at the time of the **PUT_WRITEX()** call.

Entry-sequenced files
: The record is inserted following the last record currently existing in the file.

Unstructured files

The record is inserted at the position indicated by the current value of the next-record pointer.

If a record is to be inserted into a key-sequenced or relative file and the record already exists, the **PUT_WRITEX( )** call fails, and a subsequent call to the Guardian FILE_GETINFO_ or FILEINFO procedure shows that Guardian file-system error 10 occurred.

Structured files

Inserting records into relative or entry-sequenced files

If the record is inserted into a relative or entry-sequenced file, the file must be positioned currently through its primary key. Otherwise, the **PUT_WRITEX( )** call fails, and a subsequent call to the Guardian FILE_GETINFO_ or FILEINFO procedure shows that Guardian file-system error 46 (`invalid key`) occurred.

Current-state indicators after an **PUT_WRITEX( )** call

After a successful **PUT_WRITEX( )** call, the current-state indicators for positioning mode and comparison length remain unchanged.

For key-sequenced files, the current position and the current primary-key value remain unchanged.

For relative and entry-sequenced files, the current position is that of the record just inserted and the current primary-key value is set to the value of the record's primary key.

Duplicate record found on insertion request

When you attempt to insert a record into a key-sequenced file, if a duplicate record is found, the **PUT_WRITEX( )** function returns Guardian file-system error 10 (`record already exists`) or error 71 (`duplicate record`). If the operation is part of a TMF transaction, the record is locked for the duration of the transaction.

Unstructured files

DP2 BUFFERSIZE rules

DP2 unstructured files are transparently blocked using one of the four valid DP2 blocksizes (512, 1024, 2048, or 4096 bytes; 4096 is the default). This transparent blocksize, known as BUFFERSIZE, is the transfer size used against an unstructured file. While BUFFERSIZE does not change the maximum unstructured transfer (4096 bytes), multiple I/Os can be performed to satisfy a user request depending on the BUFFERSIZE chosen. For example, if BUFFERSIZE is 512 bytes, and a request is made to write 4096 bytes, at least eight transfers, each 512 bytes long, will be made. More than eight transfers happen, in this case, if the requested transfer does not start on a BUFFERSIZE boundary.

DP2 performance with unstructured files is best when requested transfers begin on BUFFERSIZE boundaries and are integral multiples of BUFFERSIZE.

If the **PUT_WRITEX( )** call is to an unstructured disk file, data is transferred to the record location specified by the next-record pointer. The next-record pointer is updated to point to the record following the record written.

Number of bytes written

If an unstructured file is created with the odd unstructured attribute (also known as ODDUNSTR) set, the number of bytes written is exactly the number specified in *write_count*. If the odd unstructured attribute is not set when the file is created, the value of *write_count* is rounded up to an even value before the **PUT_WRITEX( )** is executed.

You set the odd unstructured attribute with the Guardian FILE_CREATE_, FILE_CREATELIST_, or CREATE procedure, or with the File Utility Program (FUP) SET and CREATE commands.

File pointers after an **PUT_WRITEX( )** call

After a successful **PUT_WRITEX( )** call to an unstructured file, the file pointers have these values:

- Current-record pointer is the next-record pointer.
- Next-record pointer is the next-record pointer plus the count written.
- End-of-file (EOF) pointer is the maximum of the EOF pointer or the next-record pointer.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **PUT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This restriction also applies to other processes that might be sharing the segment. It is the application’s responsibility to ensure this.
- If you initiated the I/O with **PUT_WRITE( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- A selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.

- You can cancel nowait I/O that was initiated with **PUT_WRITEX( )** with a call to **PUT_CANCEL( )** or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or if the Guardian AWAITIOX procedure is called with a positive time limit and specific file number and the request times out.

### Interprocess Communication Consideration

Indication that the destination process is running

If the **PUT_WRITEX( )** call is to another process, successful completion of the **PUT_WRITEX( )** call (or a Guardian AWAITIOX procedure call if nowait) indicates that the destination process is running.

## RETURN VALUES

The **PUT_WRITEX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **PUT_CANCEL(2)**, **PUT_CONTROL(2)**, **PUT_FILE_CLOSE_(2)**, **PUT_FILE_OPEN_(2)**, **PUT_LOCKFILE(2)**, **PUT_LOCKREC(2)**, **PUT_READLOCKX(2)**, **PUT_READUPDATELOCKX(2)**, **PUT_READUPDATEX(2)**, **PUT_READX(2)**, **PUT_SETMODE(2)**, **PUT_UNLOCKFILE(2)**, **PUT_UNLOCKREC(2)**, **PUT_WRITEREADX(2)**, **PUT_WRITEUPDATEUNLOCKX(2)**, **PUT_WRITEUPDATEX(2)**.

# Section 6. System Functions (r)

This section contains reference pages for Open System Services (OSS) system function calls with names that begin with **r**. These reference pages reside in the **cat2** directory and are sorted alphabetically by U.S. English conventions in this section.

## NAME

**read -** Reads from a file

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <sys/types.h>    /* optional except for POSIX.1 */
#include <unistd.h>

ssize_t read(
        int filedes,
        void *buffer,
        size_t nbytes);
```

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function, or the thread-aware **accept( )**, **dup2( )**, or **fcntl( )** function.

*buffer* Points to the buffer to receive data read.

*nbytes* Specifies the number of bytes to read from the file associated with the *filedes* parameter.

If the value of *nbytes* is 0 (zero), the **read( )** function returns 0 (zero). There are no other results.

If the value of *nbytes* is greater than **SSIZE_MAX**, the **read( )** function returns -1 and sets **errno** to [EINVAL].

## DESCRIPTION

The **read( )** function attempts to read *nbytes* bytes of data from the file associated with the *filedes* parameter into the buffer pointed to by the *buffer* parameter.

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **read( )** or **read64_( )** may be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **read64_( )** must be called.

32-bit Guardian and 64-bit OSS clients can pass 32-bit pointers and 64-bit pointers to **read64_( )**.

On regular files and devices capable of seeking, the **read( )** function starts at a position in the file given by the file pointer associated with the *filedes* parameter. Upon return from the **read( )** function, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. For such devices, the value of the file pointer after a call to the **read( )** function is always 0 (zero).

Upon successful completion, the **read( )** function returns the number of bytes actually read and placed in the buffer. This number is never greater than the value of the *nbytes* parameter.

The value returned can be less than *nbytes* if the number of bytes left in the file is less than *nbytes*, if the **read( )** request was interrupted by a signal, or if the file is a pipe, FIFO file, socket, or special file and has fewer than *nbytes* bytes immediately available for reading. For example, a **read( )** from a file associated with a terminal might return one typed line of data.

No data transfer occurs past the current end-of-file (EOF). If the starting position is at or after the end-of-file, 0 (zero) is returned.

If a **write( )** or **writev( )** call contains so much data that the file system needs to resize a pipe or FIFO buffer, a read from that pipe or FIFO file can return up to 52 kilobytes of data, regardless of the size of **PIPE_BUF**. If the buffer cannot be resized for the write operation, a read from the pipe or FIFO file does not return more than 8192 bytes per call, regardless of the setting of **O_NONBLOCK**.

When attempting to read from an empty pipe (or FIFO file):

- If no process has the pipe open for writing, the **read( )** function returns the value 0 (zero) to indicate EOF.
- If some process has the pipe open for writing:
  - If the **O_NONBLOCK** flag is not set, the **read( )** function blocks until either some data is written or the pipe is closed by all processes that had opened the pipe for writing.
  - If the **O_NONBLOCK** flag is set, the **read( )** function returns the value -1 and sets **errno** to [EAGAIN].

When attempting to read from a socket and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **read( )** function blocks until data becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **read( )** function returns the value -1 and sets **errno** to [EWOULDBLOCK].

When attempting to read from a character special file that supports nonblocking reads, such as a terminal, and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **read( )** function blocks until data becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **read( )** function returns the value -1 and sets **errno** to [EAGAIN].

If the **read( )** function is interrupted by a signal before it reads any data, it returns the value -1 with **errno** set to [EINTR]. If the **read( )** function is interrupted by a signal after it has successfully read some data, it returns the number of bytes read.

The **read( )** function returns the number of bytes with the value 0 (zero) for any unwritten portion of a regular file prior to EOF.

When reading from a device special file, the return of EOF has no effect on subsequent calls to the **read( )** function. When modem disconnect is detected, an EOF is returned. The **errno** variable is not set to [EIO].

Upon successful completion, the **read( )** function marks the **st_atime** field of the file for update.

### Use on Guardian Objects

After a call to the **fork( )**, **tdm_fork( )**, or **tdm_spawn( )** function, the initial position within a Guardian EDIT file (a file in **/G** with file code 101) is the same for both parent and child processes. However, the position is not shared. Moving the current position from within one process does not move it in the other process.

### Use From a Threaded Application

The thread-aware **read( )** function behaves exactly the same as **spt_readz( )** in the Standard POSIX Threads library. For file descriptors for regular files, if this thread-aware **read( )** function must wait for an I/O operation to complete on an open file, this function blocks the thread (instead of the entire process) that called it, while it waits for the I/O operation to complete.

This function serializes file operations on an open file. If a thread calls **read( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

## NOTES

To use the **read( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_readz(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the **read( )** function returns the number of bytes actually read and placed into the buffer. The function guarantees to read the number of bytes requested only if the descriptor references a regular file that has at least that number of bytes left before EOF.

If the read otherwise fails, the value -1 is returned, **errno** is set to indicate the error, and the contents of the buffer pointed to by the *buffer* parameter are indeterminate.

## ERRORS

If any of these conditions occurs, the **read( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor, and the process would be delayed in the read operation.

The **O_NONBLOCK** flag is set, and no data was available.

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function is in progress on a regular file and a function that is process-blocking for regular files attempts to begin an I/O operation on the same open file.

If the **read( )** function is thread-aware, the [EALREADY] value is not returned.

[EBADF] The *filedes* parameter is not a valid file descriptor open for reading.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The *buffer* parameter points to a location outside of the allocated address space of the process.

[EFILEBAD] An attempt was made to read from a Guardian EDIT file (a file in **/G** with file code 101) with a corrupted internal structure.

[EINTR] A **read( )** operation was interrupted by a signal before any data arrived.

[EINVAL] The value of the *nbytes* parameter is greater than **SSIZE_MAX**.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to read from its controlling terminal, the process is ignoring or blocking the **SIGTTIN** signal, or the process group is orphaned.
- A physical I/O error occurred. Data might have been lost during a transfer.

[EISDIR] A **read( )** operation was attempted against a directory.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] The socket is no longer connected to a peer socket.

[EOVERFLOW]
The file is a regular file, the value of *nbyte* is greater than 0 (zero), the starting position is before the End-of-File (EOF), and the starting position is greater than or equal to the file offset maximum established when the file described by *filedes* was opened.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWOULDBLOCK]
The process attempted an operation on a socket for which **O_NONBLOCK** is set, there is no data, and no error has occurred.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **creat(2)**, **creat64(2)**, **dup(2)**, **fcntl(2)**, **ioctl(2)**, **lseek(2)**, **lseek64(2)**, **open(2)**, **open64(2)**, **opendir(3)**, **pipe(2)**, **socket(2)**, **spt_readz(2)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- The value of the file pointer returned for a device that is incapable of seeking is always 0 (zero).
- When reading from a device special file, the return of EOF has no effect on subsequent calls to the **read( )** function.
- Specifying a value for the *nbytes* parameter that is greater than **SSIZE_MAX** causes the **read( )** function to return -1 and set **errno** to [EINVAL].
- **errno** can be set to [EIO] if a physical I/O error occurs.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [ECONNRESET], [EFAULT], [EFILEBAD], [EINVAL], [EISDIR], [EISGUARDIAN], [ENETDOWN], [ENOTCONN], [ETIMEDOUT], and [EWRONGID] can be returned.

The use of this function with the POSIX User Thread Model library conforms to industry standards as follows:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface
- When a signal arrives during a call to a thread-aware **read( )** function, the thread-aware **read( )** retries the I/O operation instead of returning the **errno** value [EINTR] with the following exception. If the thread-aware **fork( )** function is called by a signal handler that is running on a thread performing a thread-aware **read( )** call, the thread-aware **read( )** call in the child process returns [EINTR] to the application.

## NAME

**read64_ -** Reads from a file

## LIBRARY

H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#include <sys/types.h>** /* optional except for POSIX.1 */
**#include <unistd.h>**

**long long read64_(**
  **int** *filedes***,**
  **void _ptr64 ***buffer***,**
  **unsigned long long** *nbytes***);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*buffer* Points to the buffer to receive data read.

*nbytes* Specifies the number of bytes to read from the file associated with the *filedes* parameter.

If the value of *nbytes* is 0 (zero), the **read64_( )** function returns 0 (zero). There are no other results.

If the value of *nbytes* is greater than **SSIZE_MAX**, the **read64_( )** function returns -1 and sets **errno** to [EINVAL].

## DESCRIPTION

The **read64_( )** function attempts to read *nbytes* bytes of data from the file associated with the *filedes* parameter into the buffer pointed to by the *buffer* parameter.

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **read( )** or **read64_( )** may be called with a 32-bit pointer argument.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **read64_( )** must be called with a 64-bit pointer argument.

32-bit Guardian and 64-bit OSS clients can pass 32-bit pointers and 64-bit pointers to **read64_( )**.

On regular files and devices capable of seeking, the **read64_( )** function starts at a position in the file given by the file pointer associated with the *filedes* parameter. Upon return from the **read64_( )** function, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. For such devices, the value of the file pointer after a call to the **read64_( )** function is always 0 (zero).

Upon successful completion, the **read64_( )** function returns the number of bytes actually read and placed in the buffer. This number is never greater than the value of the *nbytes* parameter.

The value returned can be less than *nbytes* if the number of bytes left in the file is less than *nbytes*, if the **read64_( )** request was interrupted by a signal, or if the file is a pipe, FIFO file, socket, or special file and has fewer than *nbytes* bytes immediately available for reading. For example, a **read64_( )** from a file associated with a terminal might return one typed line of data.

No data transfer occurs past the current end-of-file (EOF). If the starting position is at or after the end-of-file, 0 (zero) is returned.

If a **write( )**, **write64_( )**, or **writev( )** call contains so much data that the file system needs to resize a pipe or FIFO buffer, a read from that pipe or FIFO file can return up to 52 kilobytes of data, regardless of the size of **PIPE_BUF**. If the buffer cannot be resized for the write operation, a read from the pipe or FIFO file does not return more than 8192 bytes per call, regardless of the setting of **O_NONBLOCK**.

When attempting to read from an empty pipe (or FIFO file):

- If no process has the pipe open for writing, the **read64_( )** function returns the value 0 (zero) to indicate EOF.
- If some process has the pipe open for writing:
  - If the **O_NONBLOCK** flag is not set, the **read64_( )** function blocks until either some data is written or the pipe is closed by all processes that had opened the pipe for writing.
  - If the **O_NONBLOCK** flag is set, the **read64_( )** function returns the value -1 and sets **errno** to [EAGAIN].

When attempting to read from a socket and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **read64_( )** function blocks until data becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **read64_( )** function returns the value -1 and sets **errno** to [EWOULDBLOCK].

When attempting to read from a character special file that supports nonblocking reads, such as a terminal, and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **read64_( )** function blocks until data becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **read64_( )** function returns the value -1 and sets **errno** to [EAGAIN].

If the **read64_( )** function is interrupted by a signal before it reads any data, it returns the value -1 with **errno** set to [EINTR]. If the **read64_( )** function is interrupted by a signal after it has successfully read some data, it returns the number of bytes read.

The **read64_( )** function returns the number of bytes with the value 0 (zero) for any unwritten portion of a regular file prior to EOF.

When reading from a device special file, the return of EOF has no effect on subsequent calls to the **read64_( )** function. When modem disconnect is detected, an EOF is returned. The **errno** variable is not set to [EIO].

Upon successful completion, the **read64_( )** function marks the **st_atime** field of the file for update.

### Use on Guardian Objects

After a call to the **fork( )**, **tdm_fork( )**, or **tdm_spawn( )** function, the initial position within a Guardian EDIT file (a file in **/G** with file code 101) is the same for both parent and child processes. However, the position is not shared. Moving the current position from within one process does not move it in the other process.

## RETURN VALUES

Upon successful completion, the **read64_( )** function returns the number of bytes actually read and placed into the buffer. The function guarantees to read the number of bytes requested only if the descriptor references a regular file that has at least that number of bytes left before EOF.

If the read otherwise fails, the value -1 is returned, **errno** is set to indicate the error, and the contents of the buffer pointed to by the *buffer* parameter are indeterminate.

## ERRORS

If any of these conditions occurs, the **read64_( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor, and the process would be delayed in the read operation.

The **O_NONBLOCK** flag is set, and no data was available.

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function (such as **spt_writez( )**) is in progress on a regular file and a function that is process-blocking for regular files (such as **read( )**, **spt_read( )**, or **spt_readx( )**) attempts to begin an I/O operation on the same open file.

[EBADF] The *filedes* parameter is not a valid file descriptor open for reading.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The *buffer* parameter points to a location outside of the allocated address space of the process.

[EFILEBAD] An attempt was made to read from a Guardian EDIT file (a file in **/G** with file code 101) with a corrupted internal structure.

[EINTR] A **read64_( )** operation was interrupted by a signal before any data arrived.

[EINVAL] The value of the *nbytes* parameter is greater than **SSIZE_MAX**.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to read from its controlling terminal, the process is ignoring or blocking the **SIGTTIN** signal, or the process group is orphaned.
- A physical I/O error occurred. Data might have been lost during a transfer.

[EISDIR] A **read64_( )** operation was attempted against a directory.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] The socket is no longer connected to a peer socket.

[EOVERFLOW]
The file is a regular file, the value of *nbyte* is greater than 0 (zero), the starting position is before the End-of-File (EOF), and the starting position is greater than or equal to the file offset maximum established when the file described by *filedes* was opened.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWOULDBLOCK]
The process attempted an operation on a socket for which **O_NONBLOCK** is set, there is no data, and no error has occurred.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **creat(2)**, **creat64(2)**, **dup(2)**, **fcntl(2)**, **ioctl(2)**, **lseek(2)**, **lseek64(2)**, **open(2)**, **open64(2)**, **opendir(3)**, **pipe(2)**, **read(2)**, **socket(2)**.

## STANDARDS CONFORMANCE

This API is an HP extension and is not standards conformant.

## NAME

**readlink -** Reads the value of a symbolic link

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <unistd.h>**

**int readlink(**
**const char ***_path_**,**
**char ***_buffer_**,**
**size_t** _buf_size_**);**

## PARAMETERS

*path* Specifies the pathname of the destination file or directory.

*buffer* Points to the user's buffer. The buffer should be at least as large as the *buf_size* parameter.

*buf_size* Specifies the size of the buffer.

If the actual length of the symbolic link is greater than the value of *buf_size*, the symbolic link is truncated. The buffer specified by the *buffer* parameter contains *buf_size* bytes of the link, and the value of the *buf_size* parameter is returned as the value of the function.

If the actual length of the symbolic link is less than the value of *buf_size*, then the contents of the buffer pointed to by the *buffer* parameter beyond the returned value are undefined.

If the value of *buf_size* is 0 (zero), the contents of the buffer pointed to by the *buffer* parameter are unchanged by the function call.

## DESCRIPTION

The **readlink( )** function places the contents of the symbolic link named by the *path* parameter in *buffer*, which has size *buf_size*. If the actual length of the symbolic link is less than *buf_size*, the string copied into the buffer is null-terminated.

For a **readlink( )** function to finish successfully, the calling process must have execute (search) permission for the directory containing the link.

### Use on Guardian Objects

The **readlink( )** function cannot be used on an object in the Guardian file system (**/G**). Symbolic links cannot be created in **/G**.

### Use From the Guardian Environment

The **readlink( )** function can be used by a Guardian process when the process has been compiled using the **#define _XOPEN_SOURCE_EXTENDED 1** feature-test macro or an equivalent compiler command option.

The **readlink( )** function is one of a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian filesystem file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.

- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

**NOTES**

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

**RETURN VALUES**

Upon successful completion, the **readlink( )** function returns the number of characters placed in the buffer (not including any terminating null). If the **readlink( )** function fails, the buffer is not modified, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **readlink( )** function sets **errno** to the corresponding value:

[EACCES] Search permission is denied on a component of the pathname prefix of the *path* parameter.

[EFAULT] The *path* parameter points outside the process's allocated address space.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EINVAL] The file named by the *path* parameter is not a symbolic link.

[EIO] An I/O error occurred during a read from or write to the fileset.

[ELOOP] There were too many links encountered in translating *path*.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of the following conditions exists:

- The file named by the *path* parameter does not exist.
- The *path* parameter points to an empty string.
- The *path* parameter specifies a file on a remote HP NonStop node but communication with the remote node has been lost.

[ENOROOT] The root fileset (fileset 0) is not in the STARTED state.

[ENOTDIR] A component of the pathname prefix of the *path* parameter is not a directory.

[ENXIO] An invalid device or address was specified during an input operation on a special file. One of the following events occurred:

- A device was specified that does not exist, or a request was made beyond the limits of the device.
- The fileset containing the requestor's current working directory or root directory is not mounted. This error can occur after failure and restart of an OSS name server process until the fileset has been repaired and remounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process is not running.

## RELATED INFORMATION

Functions: **link(2)**, **lstat(2)**, **stat(2)**, **symlink(2)**, **unlink(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EFSBAD], [ENOROOT], [ENXIO], and [EOSSNOTRUNNING] can be returned.

## NAME

**readv -** Reads from a file into scattered buffers

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossesrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zossedll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yossedll**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>

int readv(
        int filedes,
        struct iovec *iov,
        int iov_count);
```

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function, or the thread-aware **accept( )**, **dup2( )**, or **fcntl( )** function.

*iov* Points to an **iovec** structure that identifies the buffers into which the data is to be placed.

*iov_count* Specifies the number of entries in the **iovec** structure pointed to by the *iov* parameter.

## DESCRIPTION

The **readv( )** function attempts to read data from the file associated with the *filedes* parameter into a set of buffers. The **readv( )** function performs the same action as the **read( )** function, but it scatters the input data into the buffers specified by the array of **iovec** structure entries pointed to by the *iov* parameter.

On regular files and devices capable of seeking, the **readv( )** function starts at a position in the file given by the file pointer associated with the *filedes* parameter. Upon return from the **readv( )** function, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. For such devices, the value of the file pointer after a call to the **readv( )** function is always 0 (zero).

Upon successful completion, the **readv( )** function returns the number of bytes actually read and placed in the buffers.

No data transfer occurs past the current end-of-file (EOF). If the starting position is at or after the end-of-file, 0 (zero) is returned.

If a **write( )** or **writev( )** call contains so much data that the file system needs to resize a pipe or FIFO buffer, a read from that pipe or FIFO file can return up to 52 kilobytes of data, regardless of the size of **PIPE_BUF**. If the buffer cannot be resized for the write operation, a subsequent read from the pipe or FIFO file does not return more than 8192 bytes per call, regardless of the setting of **O_NONBLOCK**.

When attempting to read from an empty pipe (or FIFO file):

- If no process has the pipe open for writing, the **readv( )** function returns the value 0 (zero) to indicate EOF.
- If some process has the pipe open for writing:
  - — If the **O_NONBLOCK** flag is not set, the **readv( )** function blocks until either some data is written or the pipe is closed by all processes that had opened the pipe for writing.
  - — If the **O_NONBLOCK** flag is set, the **readv( )** function returns the value -1 and sets **errno** to [EAGAIN].

When attempting to read from a socket and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **readv( )** function blocks until data becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **readv( )** function returns the value -1 and sets **errno** to [EWOULDBLOCK].

When attempting to read from a character special file that supports nonblocking reads, such as a terminal, and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **readv( )** function blocks until data becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **readv( )** function returns the value -1 and sets **errno** to [EAGAIN].

If it is interrupted by a signal before it reads any data, the **readv( )** function returns the value -1 with **errno** set to [EINTR]. If it is interrupted by a signal after it has successfully read some data, the **readv( )** function returns the number of bytes read.

When reading from a device special file, the return of EOF has no effect on subsequent calls to the **readv( )** function. When modem disconnect is detected, an EOF is returned. The **errno** variable is not set to [EIO].

Upon successful completion, the **readv( )** function marks the **st_atime** field of the file for update.

The *iov_count* parameter specifies the number of entries (buffers) in the **iovec** structure pointed to by the *iov* parameter. Each **iovec** entry specifies the base address and length of an area in memory where data should be placed. The **readv( )** function always fills a buffer completely before proceeding to the next.

The **iovec** structure is defined in the **sys/uio.h** header file and contains entries with these members:

```
caddr_t   iov_base;
int       iov_len;
```

### Use on Guardian Objects

After a call to the **fork( )**, **tdm_fork( )**, or **tdm_spawn( )** function, the initial position within a Guardian EDIT file (a file in **/G** with file code 101) is the same for both parent and child processes. However, the position is not shared; moving the current position from within one process does not move it in the other process.

### Use From a Threaded Application

The thread-aware **readv( )** function behaves exactly the same as **spt_readvz( )** in the Standard POSIX Threads library. For file descriptors for regular files, if this thread-aware **readv( )** function must wait for an I/O operation to complete on an open file, this function blocks the thread (instead of the entire process) that called it, while it waits for the I/O operation to complete.

This function serializes file operations on an open file. If a thread calls **readv( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

## NOTES

To use the **readv( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_readvz(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the **readv( )** function returns the number of bytes actually read and placed into the buffers. The function guarantees to read the number of bytes requested only if the descriptor references a regular file that has at least that number of bytes left before EOF.

If the read otherwise fails, the value -1 is returned, **errno** is set to indicate the error, and the contents of the buffers are indeterminate.

**ERRORS**

If any of these conditions occurs, the **readv( )** function sets **errno** to the corresponding value:

[EAGAIN] One of these conditions occurred:

- The **O_NONBLOCK** flag is set for the file descriptor, and the process would be delayed in the read operation.
- The **O_NONBLOCK** flag is set for the file descriptor, and no data was available.

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function is in progress on a regular file and a function that is process-blocking for regular files attempts to begin an I/O operation on the same open file.

If the **readv( )** function is thread-aware, the [EALREADY] value is not returned.

[EBADF] The *filedes* parameter is not a valid file descriptor open for reading.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The **iov_base** member of the **iovec** structure points to a location outside of the allocated address space of the process.

[EFILEBAD] An attempt was made to read from a Guardian EDIT file (a file in **/G** with file code 101) with a corrupted internal structure.

[EINTR] A **readv( )** operation was interrupted by a signal before any data arrived.

[EINVAL] One of these conditions occurred:

- The sum of the **iov_len** values in the *iov* array was negative or overflowed a data item of type **ssize_t**.
- The value of the *iov_count* parameter was less than or equal to 0 (zero) or greater than **IOV_MAX**.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to read from its controlling terminal, the process is ignoring or blocking the **SIGTTIN** signal, or the process group is orphaned.
- A physical I/O error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed. Data might have been lost during a transfer.

[EISDIR] A **readv( )** operation was attempted against a directory.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] The socket is no longer connected to a peer socket.

[EOVERFLOW]
The file is a regular file, the value of *nbyte* is greater than 0 (zero), the starting position is before the End-of-File (EOF), and the starting position is greater than or equal to the file offset maximum established when the file described by *filedes* was opened.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWOULDBLOCK]
The process attempted an operation on a socket for which **O_NONBLOCK** is set, there is no data, and no error has occurred.

[EWRONGID] One of these conditions occurred:

- The process attempted an input or output operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for use of the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **creat(2)**, **dup(2)**, **fcntl(2)**, **ioctl(2)**, **lseek(2)**, **open(2)**, **opendir(3)**, **pipe(2)**, **socket(2)**, **socketpair(2)**, **spt_readvz(2)**.

## STANDARDS CONFORMANCE

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [ECONNRESET], [EFAULT], [EFILEBAD], [EINVAL], [EISDIR], [EISGUARDIAN], [ENETDOWN], [ENOTCONN], [ETIMEDOUT], and [EWRONGID] can be returned.

The use of this function with the POSIX User Thread Model library conforms to industry standards as follows:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface
- When a signal arrives during a call to a thread-aware **readv( )** function, the thread-aware **readv( )** retries the I/O operation instead of returning the **errno** value [EINTR] with the following exception. If the thread-aware **fork( )** function is called by a signal handler that is running on a thread performing a thread-aware **readv( )** call, the thread-aware **readv( )** call in the child process returns [EINTR] to the application.

## NAME

**recv -** Receives a message from a connected socket

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t recv(
        int socket,
        void *buffer,
        size_t length,
        int flags);
```

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*buffer* Points to the buffer where the message should be written.

*length* Specifies the length in bytes of the buffer pointed to by the *buffer* parameter.

*flags* Is a value that controls message reception. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_OOB** Requests out-of-band data.

**MSG_PEEK** Peeks at an incoming message. The data is treated as unread and the next call to the **recv( )** function (or similar function) will still return this data.

## DESCRIPTION

The **recv( )** function receives messages from a connected socket.

For message-based sockets (sockets of type **SOCK_DGRAM**), the entire message must be read in one call. If a message is too long to fit in the supplied buffer and **MSG_PEEK** is not set in the *flags* parameter, the excess bytes are discarded.

For stream-based sockets (sockets of type **SOCK_STREAM**), message boundaries are ignored. For such sockets, data is returned as soon as it becomes available; no data is discarded.

If no messages are available at the socket and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **recv( )** function blocks until a message arrives. If no messages are available at the socket and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **recv( )** function fails and sets **errno** to [EWOULDBLOCK].

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **recv( )** or **recv64_( )** may be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **recv64_( )** must be called.

32-bit Guardian and 64-bit OSS clients can pass 32-bit pointers and 64-bit pointers to **recv64_( )**.

**NOTES**

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When data is available, a call to the **select( )** function indicates that the file descriptor for the socket is ready for reading.

Calling the **recv( )** function with a *flags* parameter of 0 (zero) is identical to calling the **read( )** function.

To use the **recv( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_recvx(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

A call to the thread-aware **recv( )** function with a *flags* parameter value of 0 (zero) is identical to a call to the thread-aware **read( )** function.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **recv( )** function returns the length of the received message in bytes. If no data is available and the peer socket has performed an orderly shutdown, then 0 (zero) is returned.

If the **recv( )** function call fails, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **recv( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

This error is also returned if the **recv( )** function is thread-aware and the socket becomes invalid (is closed by another thread).

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINTR] A signal interrupted the function before any data was available.

This error is also returned if the **recv( )** function is thread-aware and a signal received from the **pthread_kill( )** function is not blocked, ignored, or handled.

[EINVAL] The **MSG_OOB** value is specified in the *flags* parameter and no out-of-band data is available.

[EIO] An input or output error occurred.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time might succeed.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] A receive operation was attempted on a connection-oriented socket that is not connected.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[ETIMEDOUT]
A transmission timed out on an active connection.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the operation would block.

## RELATED INFORMATION

Functions: **fcntl(2)**, **read(2)**, **recvfrom(2)**, **recvmsg(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **spt_recvx(2)**, **write(2)**.

## STANDARDS CONFORMANCE

The HP implementation does not return the **errno** value [ENOSR].

The following are HP extensions to the XPG4 specification:

- The **errno** value [ECONNRESET] can be returned when the transport-provider process is unavailable.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**recv64_** - Receives a message from a connected socket

## LIBRARY

H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

long long recv64_(
        int socket,
        void _ptr64 *buffer,
        unsigned long long length,
        int flags);
```

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*buffer* Points to the buffer where the message should be written.

*length* Specifies the length in bytes of the buffer pointed to by the *buffer* parameter.

*flags* Is a value that controls message reception. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_OOB** Requests out-of-band data.

**MSG_PEEK** Peeks at an incoming message. The data is treated as unread and the next call to the **recv64_( )** function (or similar function) will still return this data.

## DESCRIPTION

The **recv64_( )** function receives messages from a connected socket.

For message-based sockets (sockets of type **SOCK_DGRAM**), the entire message must be read in one call. If a message is too long to fit in the supplied buffer and **MSG_PEEK** is not set in the *flags* parameter, the excess bytes are discarded.

For stream-based sockets (sockets of type **SOCK_STREAM**), message boundaries are ignored. For such sockets, data is returned as soon as it becomes available; no data is discarded.

If no messages are available at the socket and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **recv64_( )** function blocks until a message arrives. If no messages are available at the socket and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **recv64_( )** function fails and sets **errno** to [EWOULDBLOCK].

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **recv( )** or **recv64_( )** may be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **recv64_( )** must be called.

32-bit Guardian and 64-bit OSS clients can pass 32-bit pointers and 64-bit pointers to **recv64_( )**.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When data is available, a call to the **select( )** function indicates that the file descriptor for the socket is ready for reading.

Calling the **recv64_( )** function with a *flags* parameter of 0 (zero) is identical to calling the **read64_( )** function.

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the **recv64_( )** function returns the length of the received message in bytes. If no data is available and the peer socket has performed an orderly shutdown, then 0 (zero) is returned.

If the **recv64_( )** function call fails, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **recv64_( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINTR] A signal interrupted the function before any data was available.

[EINVAL] The **MSG_OOB** value is specified in the *flags* parameter and no out-of-band data is available.

[EIO] An input or output error occurred.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time might succeed.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] A receive operation was attempted on a connection-oriented socket that is not connected.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[ETIMEDOUT]
A transmission timed out on an active connection.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the operation would block.

## RELATED INFORMATION

Functions: **fcntl(2)**, **read(2)**, **read64_(2)**, **recvfrom(2)**, **recvfrom64_(2)**, **recvmsg(2)**, **recvmsg64_(2)**, **select(2)**, **send(2)**, **send64_(2)**, **sendmsg(2)**, **sendmsg64_(2)**, **sendto(2)**, **sendto64_(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **write(2)**, **write64_(2)**.

## STANDARDS CONFORMANCE

This API is an HP extension and is not standards conformant.

## NAME

**recvfrom -** Receives a message from a socket

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t recvfrom(
        int socket,
        void *buffer,
        size_t length,
        int flags,
        struct sockaddr *address,
        socklen_t *address_len);
```

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*buffer* Points to the buffer where the message should be written.

*length* Specifies the length in bytes of the buffer pointed to by the *buffer* parameter.

*flags* Is a value that controls message reception. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_OOB** Requests out-of-band data.

**MSG_PEEK** Peeks at an incoming message. The data is treated as unread and the next call to the **recvfrom( )** function (or similar function) will still return this data.

*address* Specifies either a null pointer or a pointer to a **sockaddr** structure in which the sending address is to be stored. The length and format of the address depend on the address family of the socket.

For **AF_INET** sockets, a pointer to the address structure **sockaddr_in** must be cast as a **struct sockaddr**. For **AF_INET6** sockets, a pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**. For **AF_UNIX** sockets, a pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*address_len* Points to a **socklen_t** data item, which, on input, specifies the length of the **sockaddr** structure that is pointed to by the *address* parameter, and, on return, specifies the length of the address stored.

## DESCRIPTION

The **recvfrom( )** function receives messages from a connection-oriented or connectionless socket. **recvfrom( )** is normally used with connectionless sockets because it includes parameters that permit a calling program to retrieve the source address of received data.

For message-based sockets (sockets of type **SOCK_DGRAM**), the entire message must be read in one call. If a message is too long to fit in the supplied buffer and **MSG_PEEK** is not set in the *flags* parameter, the excess bytes are discarded.

For stream-based sockets (sockets of type **SOCK_STREAM**), message boundaries are ignored. For such sockets, data is returned as soon as it becomes available; no data is discarded.

If no messages are available at the socket and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **recvfrom( )** function blocks until a message arrives. If no messages are available at the socket and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **recvfrom( )** function fails and sets **errno** to [EWOULDBLOCK].

If the *address* parameter is not a null pointer, the source address of the received message is stored in the **sockaddr** structure pointed to by the *address* parameter, and the length of this address is stored in the object pointed to by the *address_len* parameter.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the address is truncated when stored.

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **recvfrom( )** or **recvfrom64_( )** may be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **recvfrom64_( )** must be called.

32-bit Guardian and 64-bit OSS clients can pass 32-bit pointers and 64-bit pointers to **recvfrom64_( )**.

**NOTES**

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When data is available, a call to the **select( )** function indicates that the file descriptor for the socket is ready for reading.

For **AF_UNIX** Release 1 sockets and for **AF_UNIX** Release 2 sockets in compatibility mode, when the file to which a sending datagram socket is bound is ulinked or renamed, and one of the **send** set of functions is called, the receiving client's call to **recvfrom( )** returns a null address (all fields in the address are zero).

For **AF_UNIX** Release 2 sockets in portability mode, when the file to which a sending datagram socket is bound is unlinked or renamed, and one of the **send** set of functions is called, the receiving client's call to **recvfrom( )** returns the fully-qualified form of the address to which the sending socket was originally bound.

For more information about **AF_UNIX** Release 2 sockets, portability mode, and compatibility mode, see the *Open System Services Programmer's Guide*.

To use the **recvfrom( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_recvfromx(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **recvfrom( )** function returns the length of the received message in bytes. If no data is available and the peer socket has performed an orderly shutdown, then 0 (zero) is returned.

If the **recvfrom( )** function call fails, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **recvfrom( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

This error is also returned if the **recvfrom( )** function is thread-aware and the socket becomes invalid (is closed by another thread).

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINTR] A signal interrupted the function before any data was available.

This error is also returned if the **recvfrom( )** function is thread-aware and a signal received from the **pthread_kill( )** function is not blocked, ignored, or handled.

[EINVAL] The **MSG_OOB** value is specified in the *flags* parameter and no out-of-band data is available.

[EIO] An input or output error occurred.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time may succeed.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] A receive operation was attempted on a connection-oriented socket that is not connected.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[ETIMEDOUT]
A transmission timed out on an active connection.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the operation would block.

## RELATED INFORMATION

Functions: **fcntl(2)**, **read(2)**, **recv(2)**, **recvmsg(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **spt_recvfromx(2)**, **write(2)**.

## STANDARDS CONFORMANCE

The HP implementation does not return the **errno** value [ENOSR].

The following are HP extensions to the XPG4 specification:

- The **errno** value [ECONNRESET] can be returned when the transport-provider process is not available.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**recvfrom64_ -** Receives a message from a socket

## LIBRARY

H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

long long recvfrom64_(
        int socket,
        void _ptr64 *buffer,
        unsigned long long length,
        int flags,
        struct sockaddr _ptr64 *address,
        socklen_t _ptr64 *address_len);
```

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*buffer* Points to the buffer where the message should be written.

*length* Specifies the length in bytes of the buffer pointed to by the *buffer* parameter.

*flags* Is a value that controls message reception. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_OOB** Requests out-of-band data.

**MSG_PEEK** Peeks at an incoming message. The data is treated as unread and the next call to the **recvfrom64_( )** function (or similar function) will still return this data.

*address* Specifies either a null pointer or a pointer to a **sockaddr** structure in which the sending address is to be stored. The length and format of the address depend on the address family of the socket.

For **AF_INET** sockets, a pointer to the address structure **sockaddr_in** must be cast as a **struct sockaddr**. For **AF_INET6** sockets, a pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**. For **AF_UNIX** sockets, a pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*address_len* Points to a long long data item, which, on input, specifies the length of the **sockaddr** structure that is pointed to by the *address* parameter, and, on return, specifies the length of the address stored.

## DESCRIPTION

The **recvfrom64_( )** function receives messages from a connection-oriented or connectionless socket. **recvfrom64_( )** is normally used with connectionless sockets because it includes parameters that permit a calling program to retrieve the source address of received data.

For message-based sockets (sockets of type **SOCK_DGRAM**), the entire message must be read in one call. If a message is too long to fit in the supplied buffer and **MSG_PEEK** is not set in the

*flags* parameter, the excess bytes are discarded.

For stream-based sockets (sockets of type **SOCK_STREAM**), message boundaries are ignored. For such sockets, data is returned as soon as it becomes available; no data is discarded.

If no messages are available at the socket and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **recvfrom64_( )** function blocks until a message arrives. If no messages are available at the socket and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **recvfrom64_( )** function fails and sets **errno** to [EWOULDBLOCK].

If the *address* parameter is not a null pointer, the source address of the received message is stored in the **sockaddr** structure pointed to by the *address* parameter, and the length of this address is stored in the object pointed to by the *address_len* parameter.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the address is truncated when stored.

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **recvfrom( )** or **recvfrom64_( )** may be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **recvfrom64_( )** must be called.

32-bit Guardian and 64-bit OSS clients can pass 32-bit pointers and 64-bit pointers to **recvfrom64_( )**.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When data is available, a call to the **select( )** function indicates that the file descriptor for the socket is ready for reading.

When the file to which a sending datagram socket is bound is unlinked or renamed, and one of the **send** set of functions is called, the receiving client's call to **recvfrom64_( )** returns a null address (all fields in the address are zero).

When the file to which a sending datagram socket is bound is unlinked or renamed, and one of the **send** set of functions is called, the receiving client's call to **recvfrom64_( )** returns the fully-qualified form of the address to which the sending socket was originally bound.

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the **recvfrom64_( )** function returns the length of the received message in bytes. If no data is available and the peer socket has performed an orderly shutdown, then 0 (zero) is returned.

If the **recvfrom64_( )** function call fails, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **recvfrom64_( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.

- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINTR] A signal interrupted the function before any data was available.

[EINVAL] The **MSG_OOB** value is specified in the *flags* parameter and no out-of-band data is available.

[EIO] An input or output error occurred.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time may succeed.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] A receive operation was attempted on a connection-oriented socket that is not connected.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[ETIMEDOUT]
A transmission timed out on an active connection.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the operation would block.

**RELATED INFORMATION**

Functions: **fcntl(2)**, **read(2)**, **read64_(2)**, **recv(2)**, **recv64_(2)**, **recvfrom(2)**, **recvmsg(2)**, **recvmsg64_(2)**, **select(2)**, **send(2)**, **send64_(2)**, **sendmsg(2)**, **sendmsg64_(2)**, **sendto(2)**, **sendto64_(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **write(2)**, **write64_(2)**.

**STANDARDS CONFORMANCE**

This API is an HP extension and is not standards conformant.

## NAME

**recvmsg -** Receives a message from a socket using a message structure

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t recvmsg(
        int socket,
        struct msghdr *message,
        int flags);
```

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*message* Points to a **msghdr** structure containing both the buffer to store the source address and the buffers for the incoming message. The length and format of the address depend on the address family for the socket. The **msg_flags** member of the structure is ignored on input but might contain meaningful values on output. For:

**AF_INET** sockets
A pointer in **msghdr** to the address structure **sockaddr_in** must be cast as a **struct sockaddr**.

**AF_INET6** sockets
A pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**.

**AF_UNIX** sockets
A pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*flags* Is a value that controls message reception. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_OOB** Requests out-of-band data.

**MSG_PEEK** Peeks at an incoming message. The data is treated as unread, and the next call to the **recvmsg( )** function (or a similar function) will still return this data.

## DESCRIPTION

The **recvmsg( )** function receives messages from a connection-oriented or connectionless socket using the **msghdr** structure. The **recvmsg( )** function is normally used with connectionless sockets because it includes parameters that permit a calling program to retrieve the source address of the received data.

For message-based sockets (sockets of type **SOCK_DGRAM**), the entire message must be read in one call. If a message is too long to fit in the supplied buffer and **MSG_PEEK** is not set in the

*flags* parameter, the excess bytes are discarded, and **MSG_TRUNC** is set in the **msg_flags** field of the **msghdr** structure.

For stream-based sockets (sockets of type **SOCK_STREAM**), message boundaries are ignored. For such sockets, data is returned as soon as it becomes available; no data is discarded.

If no messages are available at the socket and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **recvmsg( )** function blocks until a message arrives. If no messages are available at the socket and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **recvmsg( )** function fails and sets **errno** to [EWOULDBLOCK].

In the **msghdr** structure, the **msg_name** and **msg_namelen** members specify the source address if the socket is unconnected. If the socket is connected, the **msg_name** and **msg_namelen** members are ignored. The **msg_name** member can be a null pointer if no names are desired or required. The **msg_iov** and **msg_iovlen** members describe the scatter/gather locations.

Upon successful completion of the **recvmsg( )** call, the value of the **msg_flags** member of the **msghdr** structure is the bitwise OR of zero or more of the following values:

**MSG_CTRUNC**
Control data was truncated.

**MSG_OOB** Out-of-band data was received.

**MSG_TRUNC**
Normal data was truncated.

In the **msghdr** structure, the **msg_control** and **msg_controllen** members specify the ancillary data buffer that can be used only by sockets in the **AF_UNIX** domain to receive file descriptors passed from another process on the same node. The **msg_control** member can be a null pointer if ancillary data is not desired or required. If the **msg_control** member is nonnull, on input the **msg_controllen** member contains the size of the ancillary data buffer and on output it contains the size of the received ancillary data. If, on output, the **msg_controllen** member is nonzero, the ancillary data buffer contains a **cmsghdr** structure followed by one to sixteen file descriptors.

If **recvmsg( )** is called with an ancillary data buffer and **MSG_PEEK** is set, the **msg_controllen** member is valid, but the ancillary data is not meaningful (no file descriptors are received). Ancillary data is not discarded but remains available for the next call to **recvmsg( )** where **MSG_PEEK** is set.

If **recvmsg( )** is called with an ancillary data buffer that is too small to hold the available file descriptors, **MSG_CTRUNC** is set, and the excess file descriptors are discarded.

If **recvmsg( )** is called with an ancillary data buffer and one or more of the received file descriptors are unusable (perhaps because of a device error), there is no error indication until the file descriptor is used.

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **recvmsg( )** must be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **recvmsg64_( )** must be called.

To pass a 32-bit pointer from a 64-bit OSS client, **recvmsg( )** must be called.

To pass a 64-bit pointer from a 64-bit OSS client, **recvmsg_( )** must be called.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When data is available, a call to the **select( )** function indicates that the file descriptor for the socket is ready for reading.

For **AF_UNIX** Release 1 sockets and for **AF_UNIX** Release 2 sockets in compatibility mode, when the file to which a sending datagram socket is bound is unlinked or renamed, and one of the **send** set of functions is called, the receiving client's call to **recvmsg( )** returns a null address (all fields in the address are zero).

For **AF_UNIX** Release 2 sockets in portability mode, when the file to which a sending datagram socket is bound is unlinked or renamed, and one of the **send** set of functions is called, the receiving client's call to **recvmsg( )** returns the fully-qualified form of the address to which the sending socket was originally bound.

For more information about **AF_UNIX** Release 2 sockets, portability mode, and compatibility mode, see the *Open System Services Programmer's Guide*.

For J06.07 and later J-series RVUs and H06.18 and later H-series RVUs, if a memory resource allocation error occurs while attempting this operation, the operation succeeds but the resulting file descriptor is not usable. All subsequent file operations that attempt to use the file descriptor fail with the error [EBADF].

To use the **recvmsg( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_recvmsgx(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **recvmsg( )** function returns the length of the received message in bytes. If no data is available and the peer socket has performed an orderly shutdown, 0 (zero) is returned.

If the **recvmsg( )** function call fails, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **recvmsg( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

This error is also returned if the **recvmsg( )** function is thread-aware and the socket becomes invalid (is closed by another thread).

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINTR] A signal interrupted the function before any data was available.

This error is also returned if the **recvmsg( )** function is thread-aware and a signal received from the **pthread_kill( )** function is not blocked, ignored, or handled.

[EINVAL] One of these conditions occurred:

- The **MSG_OOB** value is specified in the *flags* parameter, and no out-of-band data is available.
- The sum of the values specified for the **msg_iovlen** field of the **msghdr** structure is too large for a data item of type **ssize_t**.
- The socket belongs to the **AF_INET** or **AF_INET6** domain, and the function call requested **msg_control** data.
- The socket belongs to the **AF_UNIX** domain, and the size of **msg_controllen** is less than the size of the **cmsghdr** structure plus one file descriptor.

[EIO] An input or output error occurred.

[EMFILE] The socket is in the **AF_UNIX** domain, and processing the **cmsghdr** structure would cause the receiving process to exceed **OPEN_MAX**.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] A receive operation was attempted on a connection-oriented socket that is not connected.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
A specified value for the *flags* parameter is not supported for this socket type.

[ETIMEDOUT]
A transmission timed out on an active connection.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set), and the operation would block.

### RELATED INFORMATION

Functions: **fcntl(2)**, **recv(2)**, **recvfrom(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **socketpair(2)**, **spt_recvmsgx(2)**.

### STANDARDS CONFORMANCE

The HP implementation does not return the **errno** value [ENOSR].

HP extensions to the XPG4 specification are:

- The **errno** value [ECONNRESET] can be returned when the transport-provider process is not available.
- The **errno** value [EMFILE] can be returned.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

**NAME**

**recvmsg64_ -** Receives a message from a socket using a message structure

**LIBRARY**

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

**SYNOPSIS**

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

long long recvmsg64_(
        int socket,
        struct msghdr64 _ptr64 *message,
        int flags);
```

**PARAMETERS**

*socket* Specifies the file descriptor of the socket.

*message* Points to a **msghdr64** structure containing both the buffer to store the source address and the buffers for the incoming message. The length and format of the address depend on the address family for the socket. The **msg_flags** member of the structure is ignored on input but might contain meaningful values on output. For:

**AF_INET** sockets
A pointer in **msghdr64** to the address structure **sockaddr_in** must be cast as a **struct sockaddr**.

**AF_INET6** sockets
A pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**.

**AF_UNIX** sockets
A pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*flags* Is a value that controls message reception. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_OOB** Requests out-of-band data.

**MSG_PEEK** Peeks at an incoming message. The data is treated as unread, and the next call to the **recvmsg64_( )** function (or a similar function) will still return this data.

**DESCRIPTION**

The **recvmsg64_( )** function receives messages from a connection-oriented or connectionless socket using the **msghdr64** structure. The **recvmsg64_( )** function is normally used with connectionless sockets because it includes parameters that permit a calling program to retrieve the source address of the received data.

For message-based sockets (sockets of type **SOCK_DGRAM**), the entire message must be read in one call. If a message is too long to fit in the supplied buffer and **MSG_PEEK** is not set in the

*flags* parameter, the excess bytes are discarded, and **MSG_TRUNC** is set in the **msg_flags** field of the **msghdr64** structure.

For stream-based sockets (sockets of type **SOCK_STREAM**), message boundaries are ignored. For such sockets, data is returned as soon as it becomes available; no data is discarded.

If no messages are available at the socket and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **recvmsg64_( )** function blocks until a message arrives. If no messages are available at the socket and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **recvmsg64_( )** function fails and sets **errno** to [EWOULD-BLOCK].

In the **msghdr64** structure, the **msg_name** and **msg_namelen** members specify the source address if the socket is unconnected. If the socket is connected, the **msg_name** and **msg_namelen** members are ignored. The **msg_name** member can be a null pointer if no names are desired or required. The **msg_iov** and **msg_iovlen** members describe the scatter/gather locations.

Upon successful completion of the **recvmsg64_( )** call, the value of the **msg_flags** member of the **msghdr64** structure is the bitwise OR of zero or more of the following values:

**MSG_CTRUNC**
Control data was truncated.

**MSG_OOB** Out-of-band data was received.

**MSG_TRUNC**
Normal data was truncated.

In the **msghdr64** structure, the **msg_control** and **msg_controllen** members specify the ancillary data buffer that can be used only by sockets in the **AF_UNIX** domain to receive file descriptors passed from another process on the same node. The **msg_control** member can be a null pointer if ancillary data is not desired or required. If the **msg_control** member is nonnull, on input the **msg_controllen** member contains the size of the ancillary data buffer and on output it contains the size of the received ancillary data. If, on output, the **msg_controllen** member is nonzero, the ancillary data buffer contains a **cmsghdr** structure followed by one to sixteen file descriptors.

If **recvmsg64_( )** is called with an ancillary data buffer and **MSG_PEEK** is set, the **msg_controllen** member is valid, but the ancillary data is not meaningful (no file descriptors are received). Ancillary data is not discarded but remains available for the next call to **recvmsg64_( )** where **MSG_PEEK** is set.

If **recvmsg64_( )** is called with an ancillary data buffer that is too small to hold the available file descriptors, **MSG_CTRUNC** is set, and the excess file descriptors are discarded.

If **recvmsg64_( )** is called with an ancillary data buffer and one or more of the received file descriptors are unusable (perhaps because of a device error), there is no error indication until the file descriptor is used.

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **recvmsg( )** must be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **recvmsg64_( )** must be called.

To pass a 32-bit pointer from a 64-bit OSS client, **recvmsg( )** must be called.

To pass a 64-bit pointer from a 64-bit OSS client, **recvmsg( )** must be called.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When data is available, a call to the **select( )** function indicates that the file descriptor for the socket is ready for reading.

When the file to which a sending datagram socket is bound is unlinked or renamed, and one of the **send** set of functions is called, the receiving client's call to **recvmsg64_( )** returns a null address (all fields in the address are zero).

When the file to which a sending datagram socket is bound is unlinked or renamed, and one of the **send** set of functions is called, the receiving client's call to **recvmsg64_( )** returns the fully-qualified form of the address to which the sending socket was originally bound.

If a memory resource allocation error occurs while attempting this operation, the operation succeeds but the resulting file descriptor is not usable. All subsequent file operations that attempt to use the file descriptor fail with the error [EBADF].

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the **recvmsg64_( )** function returns the length of the received message in bytes. If no data is available and the peer socket has performed an orderly shutdown, 0 (zero) is returned.

If the **recvmsg64_( )** function call fails, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **recvmsg64_( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINTR] A signal interrupted the function before any data was available.

[EINVAL] One of these conditions occurred:

- The **MSG_OOB** value is specified in the *flags* parameter, and no out-of-band data is available.
- The sum of the values specified for the **msg_iovlen** field of the **msghdr64** structure is too large for a long long data item.

- The socket belongs to the **AF_INET** or **AF_INET6** domain, and the function call requested **msg_control** data.
- The socket belongs to the **AF_UNIX** domain, and the size of **msg_controllen** is less than the size of the **cmsghdr** structure plus one file descriptor.

[EIO] An input or output error occurred.

[EMFILE] The socket is in the **AF_UNIX** domain, and processing the **cmsghdr** structure would cause the receiving process to exceed **OPEN_MAX**.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] A receive operation was attempted on a connection-oriented socket that is not connected.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
A specified value for the *flags* parameter is not supported for this socket type.

[ETIMEDOUT]
A transmission timed out on an active connection.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set), and the operation would block.

**RELATED INFORMATION**

Functions: **fcntl(2)**, **recv(2)**, **recv64_(2)**, **recvfrom(2)**, **recvfrom64_(2)**, **recvmsg(2)**, **select(2)**, **send(2)**, **send64_(2)**, **sendmsg(2)**, **sendmsg64_(2)**, **sendto(2)**, **sendto64_(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **socketpair(2)**.

**STANDARDS CONFORMANCE**

This API is an HP extension and is not standards conformant.

## NAME

**rename -** Renames a file or directory

## LIBRARY

G-series native Guardian processes: $SYSTEM.SYS*nn*.ZCRTLSRL
G-series native OSS processes: system library
H-series and J-series native Guardian processes: $SYSTEM.ZDLL*nnn*.ZCRTLDLL
H-series and J-series OSS processes: implicit libraries

## DESCRIPTION

The C run-time library supports two variants of the **rename( )** function: **rename_oss( )** and **rename_guardian( )**. The variants support the unique file-naming conventions and structures of the OSS and Guardian file systems, respectively.

The header file maps calls to **rename( )** to the variant that matches the target compilation environment. The target environment is set with the `systype` pragma.

Explicit calls to the **rename_oss( )** and **rename_guardian( )** variants in source code are made only when the behavior of one environment is desired from the other environment.

For a description of the OSS **rename( )** function and the **rename_oss( )** function, see the **rename_oss(2)** reference page. For a description of the Guardian **rename( )** function and the **rename_guardian( )** function, see the **rename_guardian(2)** reference page.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## NAME

**rename_guardian -** Renames a file (Guardian **rename( )** function)

## LIBRARY

G-series native Guardian processes: $SYSTEM.SYS*nn*.ZCRTLSRL
G-series native OSS processes: **/G/system/sys***nn***/zcrtlsrl**
H-series and J-series native Guardian processes: $SYSTEM.ZDLL*nnn*.ZCRTLDLL
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zcrtldll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/ycrtldll**

## SYNOPSIS

```
#include <stdio.h>

int rename(
        const char *from,
        const char *to);

int rename_guardian(
        const char *from,
        const char *to);
```

## PARAMETERS

*from* Specifies the current Guardian filename of the file to be renamed.

*to* Specifies the new Guardian filename of the file to be renamed.

If the *to* parameter points to an existing file, that file is replaced by the contents of the object identified by the *from* parameter.

## DESCRIPTION

The Guardian **rename( )** function and **rename_guardian( )** function rename a file within the Guardian file system.

These functions are identical in the Guardian environment. (Refer to **Interoperability Variants** later in this reference page.) Unless otherwise noted, this reference page uses **rename( )** to refer to both the Guardian **rename( )** function and **rename_guardian( )** function.

The **rename( )** function cannot rename an open file.

### Interoperability Variants

The C run-time library supports two variants of the **rename( )** function: **rename_oss( )** and **rename_guardian( )**. The variants support the unique file-naming conventions and structures of the OSS and Guardian file systems, respectively.

The header file maps calls to **rename( )** to the variant that matches the target compilation environment. The target environment is set with the `systype` pragma.

Explicit calls to the **rename_oss( )** and **rename_guardian( )** variants in source code are made only when the behavior of one environment is desired from the other environment.

**rename_oss( )** is functionally identical to the **rename( )** function of the OSS environment. It is the same as setting `systype oss` at compile time. `systype oss` is the default setting for use of the **c89** utility in the OSS environment.

**rename_guardian( )** is functionally identical to the **rename( )** function of the Guardian environment. It is the same as setting `systype guardian` at compile time. `systype guardian` is the default setting for the C and C++ compilers in the Guardian environment.

To use the **rename_oss( )** and **rename_guardian( )** functions, specify the **_TANDEM_SOURCE** feature test macro.

**RETURN VALUES**

Upon successful completion, the **rename( )** function returns a 0 (zero). Otherwise, a nonzero value is returned and the name of the file is not changed.

**RELATED INFORMATION**

Functions: **rename(2)**, **rename_oss(2)**.

**STANDARDS CONFORMANCE**

The **rename_guardian( )** function is a HP extension to the XPG4 Version 2 specification.

## NAME

**rename_oss -** Renames a file or directory (OSS **rename( )** function)

## LIBRARY

G-series native Guardian processes: $SYSTEM.SYS*nn*.ZCRTLSRL
G-series native OSS processes: system library
H-series and J-series native Guardian processes: $SYSTEM.ZDLL*nnn*.ZCRTLDLL
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <stdio.h>**

**int rename(**
**const char ****from*,
**const char ****to***);**

**int rename_oss(**
**const char ****from*,
**const char ****to***);**

## PARAMETERS

*from* Identifies the file or directory to be renamed.

*to* Identifies the new pathname of the file or directory to be renamed.

If the *to* parameter points to an existing file or an empty directory, that file or directory is replaced by the contents of the object identified by the *from* parameter. If the *to* parameter refers to a directory that is not empty, the function exits with an error.

## DESCRIPTION

The OSS **rename( )** function and **rename_oss( )** function rename a directory or a file within a fileset.

These functions are identical in the OSS environment. (Refer to **Interoperability Variants** later in this reference page.) Unless otherwise noted, this reference page uses **rename( )** to refer to both the OSS **rename( )** function and **rename_oss( )** function.

If the *from* and *to* parameters both refer to the same existing file, the function returns successfully and performs no other action.

For the function to finish successfully, the calling process must have write and search (execute) permission for the parent directories of the entities specified by the *from* and *to* parameters. If both the *from* and *to* parameters refer to directories, write and search (execute) permission are not required on the specified directories.

The entities specified by the *from* and *to* parameters both must be of the same type (that is, both directories or both files) and must reside on the same fileset. If the entity pointed to by the *to* parameter already exists, it is first removed. In that case it is guaranteed that a link specified by *to* will exist throughout the operation. This link refers to the file specified by either the *to* or *from* parameter before the operation began.

If the final component of the *from* parameter is a symbolic link, the symbolic link (not the file or directory to which it points) is renamed. If the final component of the *to* parameter is a symbolic link, the symbolic link is destroyed.

If the *from* and *to* parameters specify directories, the following requirements exist:

- The directory specified by the *from* parameter must not be an ancestor of the directory specified by the *to* parameter. For example, the *to* pathname must not contain a pathname prefix that specifies *from*.
- The directory specified by the *to* parameter must be empty, except for the **.** (dot) and **..** (dot-dot) entries.

Upon successful completion (where a rename occurs), the function marks the **st_ctime** and **st_mtime** fields of the parent directory of each file for update.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted by the same file permissions and owner privileges as any other user ID: It has no special privileges unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

Executable files that have the PRIVSOARFOPEN privilege and that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use on Guardian Objects

The OSS **rename( )** function can be used on Guardian files (that is, files within **/G**). The OSS **rename( )** function cannot be used on directories within **/G**. The new pathname must correspond to a Guardian permanent disk file name on the same volume, and the caller must have Guardian write access to the file.

A call to rename a file in **/G** is implemented as the following sequence of Guardian procedure calls:

FILE_OPEN_ with read access and shared exclusion
FILE_RENAME_
FILE_CLOSE_

### Use From the Guardian Environment

The OSS **rename( )** function belongs to a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file-system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

### Interoperability Variants

The C run-time library supports two variants of the **rename( )** function: **rename_oss( )** and **rename_guardian( )**. The variants support the unique file-naming conventions and structures of the OSS and Guardian file systems, respectively.

The header file maps calls to **rename( )** to the variant that matches the target compilation environment. The target environment is set with the `systype` pragma.

Explicit calls to the **rename_oss( )** and **rename_guardian( )** variants in source code are made only when the behavior of one environment is desired from the other environment.

**rename_oss( )** is functionally identical to the **rename( )** function of the OSS environment. It is the same as setting `systype oss` at compile time. `systype oss` is the default setting for use of the **c89** utility in the OSS environment.

**rename_guardian( )** is functionally identical to the **rename( )** function of the Guardian environment. It is the same as setting `systype guardian` at compile time. `systype guardian` is the default setting for the C and C++ compilers in the Guardian environment.

To use the **rename_oss( )** and **rename_guardian( )** functions, specify the **_TANDEM_SOURCE** feature-test macro.

## RETURN VALUES

Upon successful completion, the **rename( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **rename( )** function sets **errno** to the corresponding value. The file or directory name remains unchanged.

[EACCES] One of the following conditions exists:

- A component of either pathname denies search permission.
- One of the directories containing *from* or *to* denies write permission.
- The **S_ISVTX** flag is set on the directory containing the file referred to by the *from* parameter. However, the calling process is not any of the following:
  - The file owner
  - The directory owner
  - A process with appropriate privileges
- The **S_ISVTX** flag is set on the directory containing an existing file referred to by the *to* parameter. However, the calling process is not any of the following:
  - The file owner
  - The directory owner
  - A process with appropriate privileges

[EBUSY] One of the following conditions occurred:

- The *to* parameter specifies a directory that exists and is one of the following:
  - **/G** or **/E**
  - A Guardian disk volume or process name in **/G** (a file with an OSS pathname of the form **/G/***vol* or **/G/***process*)
  - The root directory of a fileset
  - The **/dev** directory or the **lost+found** file for a fileset (for example, **/usr/lost+found**, where **/usr** is the mount point for a fileset)
- The *from* parameter specifies one of the following:
  - **/G** or **/E**
  - **/dev**
  - **/dev/tty** or **/dev/null**
  - **lost+found**

[EEXIST] The *to* parameter specifies an existing nonempty directory or an existing Guardian file (a file in **/G**).

[EFAULT] Either the *to* or *from* parameter is an invalid address.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EGUARDIANOPEN]
The *from* parameter specifies a regular disk file on the Guardian file system (that is, a file in **/G** or in a directory within **/G**) that is already opened in exclusive mode by Enscribe.

[EINVAL] One of the following conditions exists:

- The *from* or *to* parameter is not a well-formed directory.
- The calling process attempted to rename **.** (dot) or **..** (dot-dot).
- The *from* parameter is an ancestor of the *to* parameter.

[EISDIR] The *to* parameter specifies a directory and the *from* parameter specifies a filename that is not a directory.

[ELOOP] Too many symbolic links were encountered in translating either the *to* or *from* parameter.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *to* parameter

- The pathname pointed to by the *from* parameter
- A component of the pathname pointed to by the *to* parameter
- A component of the pathname pointed to by the *from* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *to* or *from* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of the following conditions exists:

- The path specified by the *from* parameter is an empty string.
- The file specified by the *from* parameter does not exist.
- The *path* parameter specifies a file on a remote HP NonStop node but communication with the remote node has been lost.

[ENOMEM] The system has insufficient resources to complete the operation.

[ENOROOT] One of the following conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node and communication with the remote name server has been lost.

[ENOSPC] The directory that would contain *to* cannot be extended, because the fileset is out of space.

[ENOTDIR] The *from* parameter specifies a directory and the *to* parameter specifies a file (not a directory), or a component of either path is not a directory.

[ENXIO] The fileset containing the client's current working directory or root directory is not mounted.

[EOSSNOTRUNNING]
A required system process is not running.

[EPERM] One of the following conditions exists:

- The call attempted to create a file named **lost+found** in the root directory of an OSS fileset.
- The call attempted to rename a Guardian file (that is, a file within **/G**) that is not a regular file. This error usually occurs when an attempt is made to rename a file as a Guardian subvolume or to rename a Guardian subvolume.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The requested operation requires writing in a directory on a read-only fileset.

[ETXTBSY] The file to be renamed is already busy. The file specified by the *from* parameter is a NonStop SQL/MP object file that is currently executing.

[EXDEV] The link specified by the *to* parameter and the file specified by the *from* parameter are on different filesets.

## RELATED INFORMATION

Functions: **chmod(2)**, **link(2)**, **mkdir(2)**, **rename(2)**, **rename_guardian(2)**, **rmdir(2)**, **unlink(2)**.

Commands: **chmod(1)**, **mkdir(1)**, **mv(1)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- The calling process is not required to have write or search permission for a directory in order to rename the directory.
- The **errno** value [EBUSY] is returned when either directory is in use by another process.
- The **errno** value [EMLINK] is not returned, because links to directories are not allowed.

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EFSBAD], [EGUARDIANOPEN], [ENOMEM], [ENOROOT], [ENXIO], [EOSSNOTRUNNING], [EPERM], and [EXDEV] can be returned.

The **rename_oss( )** function is a HP extension to the XPG4 Version 2 specification.

## NAME

**rmdir -** Removes a directory

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <unistd.h>**

**int rmdir(**
    **const char ***path**);**

## PARAMETERS

*path* Specifies the directory pathname. The pathname cannot be specified as **.** (dot) or **..** (dot-dot). If either value is used, the call fails and **errno** is set to [EINVAL].

The final component of the *path* parameter cannot be a symbolic link. If the final component is a symbolic link, the call fails and **errno** is set to [ENOTDIR].

## DESCRIPTION

The **rmdir( )** function removes the directory specified by the *path* parameter. The directory is removed only if it is an empty directory.

For the **rmdir( )** function to execute successfully, the calling process must have write access to the parent directory of the directory specified by the *path* parameter.

If no process has the directory open, the space occupied by the directory is freed and the directory is no longer accessible. If one or more processes have the specified directory open, the **.** (dot) and **..** (dot-dot) entries in the specified directory, if present, are removed before the **rmdir( )** function returns, and no new entries can be created in the directory. However, the directory is not removed until all references to the directory have been closed.

The **rmdir( )** function can be used to remove a root directory (/ cannot be removed) or the current working directory of a process. However, such an action has the following consequence:

- If the root directory of a process is removed, subsequent attempts by that process to resolve absolute pathnames will fail with **errno** set to [ENOENT].
- If the current working directory of a process is removed, subsequent attempts by that process to resolve relative pathnames will fail with **errno** set to [ENOENT].

If the directory specified by the *path* parameter is any of the following, the operation fails and **errno** is set to [EBUSY]:

- **/E** or **/G** (the Guardian file system)
- A disk volume or process within **/G** (**/G/***vol* or **/G/***process*)
- A mount point for a fileset
- **lost+found** in the root directory of a fileset

Upon successful completion, the **rmdir( )** function marks the **st_ctime** and **st_mtime** fields of the parent directory for update.

Because directories can have only one link, a successful call to the **rmdir( )** function always sets the link count to 0 (zero).

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted by the same file permissions and owner privileges as any other user ID: It has no special privileges unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

Executable files that have the PRIVSOARFOPEN privilege and that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use From the Guardian Environment

The **rmdir( )** function is one of a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **rmdir( )** function returns the value 0 (zero). If the **rmdir( )** function fails, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **rmdir( )** function sets **errno** to the corresponding value:

[EACCES] One of the following conditions exists:

- Search permission is denied on a component of the directory pathname specified by the *path* parameter.
- Write permission is denied on the parent directory of the directory to be removed.
- The **S_ISVTX flag** is set on the directory containing the directory referred to by the *path* parameter. However, the calling process is not any of the following:
  - — The parent directory owner

— The directory owner

— A process with appropriate privileges

[EBUSY] One of the following conditions exists:

- The directory specified by the *path* parameter is in use as the mount point for a fileset.
- The directory specified by the *path* parameter is **/E** or **/G** (the Guardian file system) or a disk volume or process within **/G** (has an OSS pathname of the form **/G/***vol* or **/G/***process*).
- The directory specified by the *path* parameter is the **lost+found** directory in the root directory for a fileset.

[EFAULT] The *path* parameter is an invalid address.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EINVAL] The specified **.** (dot) or **..** (dot-dot) pathname cannot be removed.

[EIO] During a read from or write to a fileset, an I/O error occurred.

[ELOOP] Too many symbolic links were encountered in translating the *path* parameter.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of the following conditions exists:

- The directory specified by the *path* parameter does not exist.
- The *path* parameter specifies an empty string.
- The *path* parameter specifies a file on a remote HP NonStop node but communication with the remote node has been lost.

[ENOROOT] One of the following conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node and communication with the remote name server has been lost.

[ENOTDIR] One of the following conditions exists:

- A component of the directory pathname specified by the *path* parameter is not a directory.
- The final component of the *path* parameter is a symbolic link.

[ENOTEMPTY]
The directory specified by the *path* parameter is not empty.

[ENXIO] The fileset containing the client's current working directory or root directory is not mounted.

[EOSSNOTRUNNING]
A required OSS system process is not running.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The directory specified by the *path* parameter resides on a read-only fileset.

## RELATED INFORMATION

Functions: **chmod(2)**, **chroot(2)**, **mkdir(2)**, **mkfifo(3)**, **mknod(2)**, **remove(3)**, **rename(2)**, **umask(2)**, **unlink(2)**.

Commands: **rmdir(1)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- The **rmdir( )** function can be used to remove the root directory or the current working directory of a process. The consequences of such an action are described under **DESCRIPTION**.
- The **errno** value [ENOTEMPTY] is returned instead of [EEXIST].

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EFSBAD], [EINVAL], [ENOROOT], [ENOTEMPTY], [ENXIO], and [EOSSNOTRUNNING] can be returned.

# Section 7. System Functions (s and S)

This section contains reference pages for Open System Services (OSS) system function calls with names that begin with **s** or **S**. These reference pages reside in the **cat2** directory and are sorted alphabetically by U.S. English conventions in this section.

## NAME

**sched_get_priority_max -** Returns the maximum priority for a scheduling policy

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll*nnn*/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <sched.h>**] **#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int sched_get_priority_max(**
**int** *policy***);**

## PARAMETERS

*policy* Specifies one of the scheduling policies defined in the **sched.h** header file.

## DESCRIPTION

The **sched_get_priority_max( )** function returns the maximum priority for the scheduling policy specified by the *policy* parameter. The value of *policy* must be one of the scheduling policies (**SCHED_FIFO**, **SCHED_RR**, or **SCHED_OTHER**) defined in the **sched.h** header file.

No special privileges are needed to use the **sched_get_priority_max( )** function.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll*nnn*/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

On a successful call, the requested value is returned. If a call fails, a value of -1 is returned and **errno** is set to indicate the error.

## ERRORS

The **sched_get_priority_max( )** function fails under the following condition:

[EINVAL] The value of the *policy* parameter does not represent a defined scheduling policy.

## RELATED INFORMATION

Functions: **sched_get_priority_min(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**sched_get_priority_min -** Returns the minimum priority for a scheduling policy

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <sched.h>**] **#include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int sched_get_priority_min(**
    **int** *policy***);**

## PARAMETERS

*policy* Specifies one of the scheduling policies defined in the **sched.h** header file.

## DESCRIPTION

The **sched_get_priority_min( )** function returns the minimum priority for the scheduling policy specified by the *policy* parameter. The value of *policy* must be one of the scheduling policies (**SCHED_FIFO**, **SCHED_RR**, or **SCHED_OTHER**) defined in the **sched.h** header file.

No special privileges are needed to use the **sched_get_priority_min( )** function.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

### RETURN VALUES

On a successful call, the requested value is returned. If the call fails, a value of -1 is returned and **errno** is set to indicate the error.

### ERRORS

The **sched_get_priority_min( )** function fails under the following condition:

[EINVAL] The value of the *policy* parameter does not represent a defined scheduling policy.

### RELATED INFORMATION

Functions: **sched_get_priority_max(2)**.

### STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**sched_yield -** Signals a willingness to yield the processor to another thread in the current process

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**
H-series and J-series OSS processes that use the Standard POSIX Threads library:
**/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <sched.h>] #include <pthread.h>** | **#include <spthread.h>**
/* pthread.h is required to use POSIX User Thread Model library */
/* spthread.h is required to use Standard POSIX Threads library */

**int sched_yield(void);**

## DESCRIPTION

This function forces the calling thread to relinquish its processor until it again becomes the head of its thread list. This function notifies the thread scheduler that the calling thread is willing to release its processor to other threads of equivalent or greater scheduling precedence. (A thread generally releases its processor to a thread of a greater scheduling precedence without calling this function.) If no other threads of equivalent or greater scheduling precedence are ready to execute, the calling thread continues.

This function can allow you to use knowledge of the details of an application to improve its performance. If a thread does not call **sched_yield( )**, other threads might be given the opportunity to run at arbitrary points (possibly even when the interrupted thread holds a required resource). By making strategic calls to **sched_yield( )**, other threads can be given the opportunity to run when the resources are free, which can sometimes improve performance by reducing contention for resources.

Consider calling this function after a thread has released a resource (such as a mutex) that is heavily used by other threads. This call can be especially important if the thread acquires and releases the resource inside a tight loop.

Use this function carefully and sparingly, because misuse can cause unnecessary context switching, which increases overhead and degrades performance. For example, performance is degraded if a thread yields while it holds a resource needed by the threads it is yielding to. Likewise, yielding is pointless unless another thread is ready to run.

On systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs, you can use either the POSIX User Thread Model library or the Standard POSIX Threads library for threaded applications.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use the POSIX User Thread Model library with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the POSIX User Thread Model library on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Include the **pthread.h** header file in the application.
- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

Upon successful completion, this function returns a 0 (zero).

## RELATED INFORMATION

Functions: **pthread_attr_setschedparam(2)**, **pthread_setschedparam(2)**.

## STANDARDS CONFORMANCE

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

**NAME**

**select -** Selects among file descriptors for synchronous input/output multiplexing

**LIBRARY**

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

**SYNOPSIS**

**#include <sys/time.h>**

**int select(**
    **int** *nfds***,**
    **fd_set ***readfds***,**
    **fd_set ***writefds***,**
    **fd_set ***errorfds***,**
    **struct timeval ***timeout***);**

**void FD_CLR(**
    **int** *fd***,**
    **fd_set ***fdset***);**

**int FD_ISSET(**
    **int** *fd***,**
    **fd_set ***fdset***);**

**void FD_SET(**
    **int** *fd***,**
    **fd_set ***fdset***);**

**void FD_ZERO(**
    **fd_set ***fdset***);**

**PARAMETERS**

*nfds* Specifies the range of open file descriptors that might be ready for reading or writing or that have exceptions pending. The **select( )** function tests file descriptors in the range of 0 (zero) through *nfds* -1.

The *nfds* parameter cannot be greater than **FD_SETSIZE**.

*readfds* Points to a file descriptor set consisting of file descriptors of objects opened for reading. When the function is called, this file descriptor set specifies file descriptors to be checked for being ready to read. Upon return from a successful call, this file descriptor set specifies file descriptors that are ready to be read.

*writefds* Points to a file descriptor set consisting of file descriptors for objects opened for writing. When the function is called, this file descriptor set specifies file descriptors to be checked for being ready to write. Upon return from a successful call, this file descriptor set specifies file descriptors that are ready to be written.

*errorfds* Points to a file descriptor set consisting of file descriptors for objects opened for reading or writing. When the function is called, this file descriptor set specifies file descriptors to be checked for having exception conditions pending. Upon return from a successful call, this file descriptor set specifies file descriptors that have exception conditions pending.

*timeout* Points to a type **timeval** structure that specifies the time to wait for a response from a call to the **select( )** function. When the *timeout* parameter is not a null pointer, the maximum time interval to wait for the **select( )** function to finish is specified by values stored in space reserved by the type **timeval** structure pointed to by the *timeout* parameter. A *timeout* value of 0 (zero) is treated as "do not wait". A *timeout* value of less than 10 milliseconds is rounded up so that the *timeout* value used is at least 10 milliseconds. A *timeout* value more than 10 milliseconds is rounded down to the nearest multiple of 10 milliseconds.

The object pointed to by the *timeout* parameter can be modified after successful completion of the call.

*fd* Specifies a file descriptor.

*fdset* Points to a file descriptor set.

## DESCRIPTION

The **select( )** function checks the status of objects identified by bit masks called file descriptor sets.

Each file descriptor set consists of an array of bits whose relative position and state represent a file descriptor and the true or false status for the condition of its corresponding object. An object is an open file descriptor for an OSS directory (that is, a directory that is not in **/G** or **/E**), a socket, a regular file, a terminal device file, a pipe, or a FIFO.

There is a file descriptor set for reading, for writing, and for pending exceptions. The *readfds*, *writefds*, and *errorfds* parameters point to these file descriptor sets.

When the **select( )** function is called, it checks the file descriptor sets in the range 0 through *nfds*-1. If any file descriptors are ready for reading or writing, or have a pending exception, the **select( )** function returns a modified file descriptor set.

If no condition is true for any specified file descriptor in any specified file descriptor set, the **select( )** function blocks until one of these conditions occurs:

- A specified condition is true for one of the specified descriptors in one of the specified sets.
- The interval specified by the *timeout* parameter elapses. If the *timeout* parameter points to a structure whose members have the value 0 (zero), process blocking does not occur.

A modified file descriptor set has these characteristics:

- It is a selected file descriptor set pointed to by the *readfds*, *writefds*, and *errorfds* parameters.
- When the function was called, the file descriptor set had at least one bit set that corresponded to an active file descriptor.
- The object represented by the set bit is any of these:

— is ready for reading

— is ready for writing

— has an exception pending

When these conditions exist, a corresponding bit position is set in the returned file descriptor set pointed to by the *readfds*, *writefds*, and *errorfds* parameters.

On return, the **select( )** function replaces the original file descriptor sets with the corresponding file descriptor sets that have a bit set for each file descriptor representing those objects that are ready for the requested operation. The total number of ready objects represented by set bits in all the file descriptor sets is returned by the **select( )** function.

After a file descriptor set is created, it can be modified with these macros:

**FD_CLR(***fd***, &***fdset***)**
Clears the file descriptor bit specified by the *fd* parameter in the file descriptor set pointed to by the *fdset* parameter.

**FD_ISSET(***fd***, &***fdset***)**
Returns a nonzero value when the file descriptor bit specified by the *fd* parameter is set in the file descriptor set pointed to by the *fdset* parameter. Otherwise, the value 0 (zero) is returned.

**FD_SET(***fd***, &***fdset***)**
Includes the particular file descriptor bit specified by the *fd* parameter in the file descriptor set pointed to by the *fdset* parameter.

**FD_ZERO(&***fdset***)**
Initializes the file descriptor set pointed to by the *fdset* parameter to a null value.

The behavior of these macros is undefined when the *fd* parameter has a value less than 0 (zero) or greater than or equal to **FD_SETSIZE**.

**Use on Guardian Objects**

You can use the **select( )** function on regular files (disk files) or EDIT files in **/G**. Such files are always ready for selection.

You can use the **select( )** function on an OSS terminal (Telserv or OSSTTY). You cannot use **select( )** function on any other type of Guardian object.

If **select( )** is called using a file descriptor for a version of the Telserv process or OSSTTY process that does not support **select( )**, the call fails, and **errno** is set to the value of [ENOTSUP].

**Use From the Guardian Environment**

The **select( )** function is one of a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.

- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.

- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

### Use From a Threaded Application

The thread-aware **select( )** function can be used to check the status of multiple file descriptors. To check the status of a single file descriptor, use the **_PUT_SELECT_SINGLE_** feature test macro, which uses another thread-aware version of the **select( )** function and provides better performance. To use **_PUT_SELECT_SINGLE_**, you must include the **pthread.h** header file in the application and link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

## NOTES

Beginning with the release of product version T9055G12 and product version update (PVU) T8645G08 AAO, **FD_SETSIZE** was increased. In T8645G08 AAO, the **FD_SETSIZE** literal in the **sys/time.h** file was increased from 1024 to 4096. Object modules that were compiled using pre-T8645G08-AAO header files use the smaller 1024 **FD_SETSIZE** and are termed *old objects*. Object modules that were compiled using T8645G08 AAO header files use the bigger 4096 **FD_SETSIZE** and are termed *new objects*. The way an application behaves for old or new objects depends on the way in which it calls the **select( )** function:

- The application can use a variable value for the *nfds* parameter that is based on the highest numbered file descriptor; for example:

  ```
  fd = open(... .);
  err = select(fd + 1, ...);
  ```

- The application can use a fixed value for the *nfds* parameter that is based on the value of the **FD_SETSIZE** literal at the time of compilation; for example:

  ```
  err = select(FD_SETSIZE, ...);
  ```

Applications composed entirely of old objects that use a variable value for the *nfds* parameter run correctly on systems running pre-T9055G12 PVUs and run correctly on systems running T9055G12 or a more recent PVU. For such applications, **select( )** calls are restricted to 1024 file descriptors.

Applications composed entirely of old objects that use a fixed value for the *nfds* parameter run correctly on systems running pre-T9055G12 or T9055G12 or newer PVUs. For such applications, **select( )** calls are restricted to 1024 file descriptors.

Applications composed entirely of new objects that use a variable value for the *nfds* parameter run correctly on systems running pre-T9055G12 PVUs or T9055G12 or newer PVUs. For such applications, **select( )** calls are restricted to 1024 file descriptors under pre-T9055G12 PVUs or to 4096 file descriptors under T9055G12 or newer PVUs.

Applications composed entirely of new objects that use a fixed value for the *nfds* parameter run correctly on systems running T9055G12 or newer PVUs. For such applications, **select( )** calls are restricted to 4096 file descriptors.

Applications composed entirely of new objects that use a fixed value for the *nfds* parameter are unsafe on systems running pre-T9055G12 PVUs.

Applications that mix old objects and new objects are unsafe on any system. You must compile all object modules of an application using a consistent set of header file definitions.

Specifying arbitrarily large values for the *nfds* parameter can cause the function to behave inefficiently.

The time limit value specified by the *timeout* parameter has no effect on the operation of the **alarm( )** or **settimer( )** function.

To use the **select( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_select(2)** and **spt_select_single_np(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **select( )** function returns the number of ready objects represented by corresponding file descriptor bits in all the file descriptor sets. When an error occurs, the value -1 is returned, and **errno** is set to indicate the error.

When the time limit specified by the *timeout* parameter expires, the **select( )** function returns the value 0 (zero), and all bits in the objects pointed to by the *readfds*, *writefds*, and *errorfds* parameters are also set to 0 (zero).

When **select( )** returns an error, the file descriptor sets pointed to by the *readfds*, *writefds*, and *errorfds* parameters remain unmodified.

The **FD_CLR**, **FD_SET**, and **FD_ZERO** macros do not return values. The **FD_ISSET** macro returns a nonzero value when the bit for the file descriptor specified by its *fd* parameter is set in the file descriptor set pointed to by its *fdset* parameter; otherwise, the **FD_ISSET** macro returns 0 (zero).

**ERRORS**

If any of these conditions occur, the **select( )** function sets **errno** to the corresponding value:

[EBADF] One of the specified file descriptor sets is invalid. One of these conditions occurred:

- The invalid file descriptor set contains a file descriptor for a file that is not open.

- The invalid file descriptor set contains a file descriptor that identifies an **AF_INET AF_INET6** socket, but the current processor is not running a transport agent process to support the socket. The file descriptor can only be closed.

This error is also returned if the **select( )** function is thread-aware and the socket becomes invalid (is closed by another thread).

[EINTR] A signal was delivered before the time limit specified by the *timeout* parameter expired and before any of the selected events occurred.

This error is also returned if the **select( )** function is thread-aware and a signal received from the **pthread_kill( )** function is not blocked, ignored, or handled.

[EINVAL] One of these conditions occurred:

- The value specified for the *nfds* parameter is less than 0 (zero) or greater than **FD_SETSIZE**.
- The time limit specified by the *timeout* parameter is invalid. One of its components is negative or too large.

[ENETDOWN]
One of the specified file descriptors specifies a file on a remote node, but communication with the remote node has been lost.

[ENOTSUP] One of the specified file descriptors specifies a terminal device that does not support **select( )**. Only terminal devices on systems running G06.27 and later G-series RVUs and H06.05 and later H-series RVUs support the **select( )** function.

[ETHNOTRUNNING]
One of the specified file descriptors is a terminal device file descriptor and the OSS terminal helper process is not running in the same processor as the application. Under normal conditions, the OSS terminal helper process runs in all processors. If this error is returned, contact your service provider and provide a copy of the Event Management Service (EMS) log. If your local operating procedures require contacting the Global Customer Support Center (GCSC), supply your system number and the numbers and versions of all related products as well.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Functions: **fcntl(2)**, **read(2)**, **spt_select(2)**, **spt_select_single_np(2)**, **write(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** value [ENOTSUP] can be returned for a call that attempts to select a terminal device file for a terminal process that does not support **select( )**. Only terminal devices on systems running G06.27 and later G-series RVUs and H06.05 and later H-series RVUs support the **select( )** function.
- The **errno** value [ENETDOWN] can be returned.

- The **errno** value [ETHNOTRUNNING] can be returned.
- The time interval specified by the *timeout* parameter must meet these criteria:
  - The maximum interval is 2**31 seconds plus 2**31 microseconds. If a value greater than this is specified, the maximum is used instead.
  - If a specified interval is not a whole multiple of 10 milliseconds, the next highest whole multiple is used instead.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**semctl -** Performs semaphore control operations

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/yosskdll**

## SYNOPSIS

```
#include <sys/sem.h>

int semctl(
        int semid,
        int semnum,
        int cmd [,
        . . . ]);
```

In this instance, the elipsis ( . . . ) indicates that the function is extensible. An additional, optional parameter can be specified.

## PARAMETERS

*semid* Specifies the ID of the semaphore set.

*semnum* Specifies the number of the semaphore to be processed.

*cmd* Specifies the type of operation (see **DESCRIPTION**).

the fourth parameter

Is defined in the XPG4 specification in a manner that avoids conflict with the ISO C standard.

This parameter is required when the *cmd* parameter has values of **GETALL**, **IPC_SET**, **IPC_STAT**, **SETALL**, and **SETVAL**. The fourth parameter can be omitted in all other calls.

This parameter must be defined in a user program as follows:

```
union semun {
        int                 val;
        struct semid_ds     *buf;
        unsigned short      *array;
} arg;
```

The fields have the following definitions:

*val* Contains the semaphore value to which the **semval** field of the **sem** structure is set when the *cmd* parameter has the value **SETVAL**. Individual semaphores are defined using the **sem** structure, where **semval** is one of the structure's fields.

**buf* Points to a **semid_ds** structure. When the **IPC_STAT** value is specified for the *cmd* parameter, **semctl( )** copies the contents of the requested **semid_ds** structure into the location pointed to by the **buf* parameter.

When the **IPC_SET** value is specified for the *cmd* parameter, **semctl( )** copies the contents of the location pointed to by the **buf* parameter into the **semid_ds** structure associated with the semaphore specified by the *semnum* parameter.

| | |
|---|---|
| **array* | Points to an array of semaphore values. These values are returned when the *cmd* parameter has the value **GETALL**.<br>These values are used to set semaphore values when the *cmd* parameter has the value **SETALL**. |
| *arg* | Specifies the instance of the union used for the fourth parameter. |

**DESCRIPTION**

An OSS semaphore is identified by a set ID and by a unique semaphore number within that set ID. The semaphore set ID is unique within an HP NonStop server node.

The **semctl( )** function allows a process to perform various operations on the following:

- An individual semaphore within a semaphore set
- All semaphores within a semaphore set
- The **semid_ds** structure associated with the semaphore set

The **semctl( )** function also allows a process to remove the semaphore set ID and its associated **semid_ds** structure. Individual semaphores are defined using the **sem** structure.

The *cmd* parameter determines which operation is performed. The following values for *cmd* operate on the specified semaphore (given by the *semnum* parameter) within the specified semaphore set (given by the *semid* parameter):

**GETNCNT** Returns the number of processes waiting for the specified semaphore's value to become greater than its current value. This number is returned as the value of the **semncnt** field from the **semid_ds** structure.

This operation requires read access permission.

**GETPID** Returns the OSS process ID of the process that last operated on the specified semaphore. This operation requires read access permission.

**GETVAL** Returns the value of the specified semaphore. This operation requires read access permission.

**GETZCNT** Returns the number of processes waiting for the specified semaphore's value to become 0 (zero). This number is returned as the value of the **semzcnt** field from the **semid_ds** structure.

This operation requires read access permission.

**SETVAL** Sets the value of the specified semaphore to the value specified through the fourth parameter (*arg.val*). When this operation successfully executes, the system clears the semaphore's adjust-on-exit value in all processes that have a **semadj** value for this semaphore. It also wakes up all processes that are waiting on this semaphore when the value of **semzcnt** or **semncnt** is greater than zero, depending on whether the value of this semaphore is set to zero or a positive integer respectively.

This operation requires alter access permission.

The following values for *cmd* operate on all the semaphores in the specified semaphore set:

**GETALL** Returns the values of all semaphores in the set by placing these values in the array pointed to in fourth parameter (*arg.array*). This operation requires read access permission.

**SETALL** Sets the respective values of all semaphores in the set to the values specified in the array pointed to in the fourth parameter (*arg.array*). When this operation successfully executes, the system clears the semaphore's adjust-on-exit value in all processes that have a **semadj** value for this semaphore. It also wakes up all processes that are waiting on a semaphore when the value of **semzcnt** or **semncnt** is greater than zero, depending on whether the respective value of a specific semaphore is set to zero or a positive integer respectively.

This operation requires alter access permission.

The following interprocess communications (IPC) commands can also be used as values for *cmd*:

**IPC_RMID** Removes the semaphore set ID and destroys the set of semaphores and the **semid_ds** structure associated with it.

This is a restricted operation. The effective user ID of the calling process must be either the super ID or equal to the value of the **sem_perm.cuid** or **sem_perm.uid** field in the associated **semid_ds** structure.

**IPC_SET** Sets the semaphore set by copying selected user-supplied values in the structure pointed to in the fourth parameter (*arg.buf*) into corresponding fields in the **semid_ds** structure associated with the semaphore set ID.

This is a restricted operation. The calling process must either have appropriate privileges, be the process that created the semaphore set, or be the process that currently owns the semaphore set.

The fields are set as follows:

- The **sem_perm.uid** field is set as specified in the **uid** field of the **semid_ds ipc_perm** structure pointed to in the fourth parameter (*arg.buf*).
- The **sem_perm.gid** field is set as specified in the **gid** field of the **semid_ds ipc_perm** structure pointed to in the fourth parameter (*arg.buf*).
- The **sem_perm.mode** field is set to the access modes for the semaphore set. Only the low-order nine bits are set.
- The **sem_ctime** field is updated.

**IPC_STAT** Queries the semaphore set ID by copying the contents of its associated **semid_ds** structure into the structure pointed to in the fourth parameter (*arg.buf*). This operation requires read access permission.

### Use From the Guardian Environment

If called from a Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

## RETURN VALUES

Upon successful completion, the value returned depends on the value of the *cmd* parameter as follows:

**GETNCNT** Returns the value of the **semncnt** field from the **semid_ds** structure.

**GETPID** Returns the value of the **sempid** field from the **semid_ds** structure.

**GETVAL** Returns the value of the **semval** field from the **semid_ds** structure.

**GETZCNT** Returns the value of the **semzcnt** field from the **semid_ds** structure.

Upon successful completion, all other values of *cmd* return the value 0 (zero).

If the **semctl( )** function fails, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **semctl( )** function sets **errno** to the corresponding value:

[EACCES] The calling process does not have the required read or alter access.

[EFAULT] One of the following is true:

- The *cmd* parameter is **IPC_STAT**, and either the structure pointed to in the fourth parameter (*arg.buf*) is not in the address space of the process or the function cannot write into the structure pointed to in the fourth parameter.
- The *cmd* parameter is **GETALL**, and either the structure pointed to in the fourth parameter (*arg.buf*) is not in the address space of the process or the function cannot write into the structure pointed to in the fourth parameter.

[EINVAL] One of the following conditions is true:

- The *semid* parameter is not a valid semaphore set ID.
- The value of the *semnum* parameter is less than 0 (zero), or it is greater than or equal to the value of the **sem_nsems** field in the **semid_ds** structure.
- The *cmd* parameter is not a valid operation.

[ENOTOSS] The calling process is not an OSS process. The requested operation is not supported from the Guardian environment.

[EPERM] All of the following conditions are true:

- The *cmd* parameter is equal to **IPC_RMID** or **IPC_SET**.
- The effective user ID of the calling process does not have appropriate privileges.
- The effective user ID of the calling process is not equal to the value of the **sem_perm.cuid** or **sem_perm.uid** field in the **semid_ds** structure associated with the semaphore set ID.

[ERANGE] The *cmd* parameter is **SETALL** or **SETVAL**, and the semaphore value in the **semval** field of the **semid_ds** structure associated with the semaphore set ID is greater than the system-defined maximum.

### RELATED INFORMATION

Commands: **ipcrm(1)**, **ipcs(1)**.

Functions: **ftok(3)**, **semget(2)**, **semop(2)**.

### STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT] and [ENOTOSS] can be returned.

## NAME

**semget -** Creates a new semaphore set ID or returns the ID of an existing semaphore set

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

## SYNOPSIS

**#include <sys/sem.h>**

**int semget(**
    **key_t** *key***,**
    **int** *nsems***,**
    **int** *semflg***);**

## PARAMETERS

*key* Specifies the key that identifies the semaphore set. The **IPC_PRIVATE** key can be used to ensure the return of a new (unused) semaphore set ID in the semaphore set table.

*nsems* Specifies the number of OSS semaphores to create in the semaphore set.

*semflg* Specifies the creation flags. Possible values are as follows:

**IPC_CREAT** If the key does not exist, the **semget( )** function creates a semaphore set ID using the given key.

**IPC_CREAT | IPC_EXCL**
If the key already exists, the **semget( )** function fails and returns an error notification.

## DESCRIPTION

The **semget( )** function returns the semaphore set ID for the semaphore set identified by the *key* parameter. If the *key* parameter already has a semaphore set ID associated with it and (*semflg* **& IPC_CREAT**) is 0 (zero), that ID is returned.

A new semaphore set ID, the associated semaphore set table, and a new semaphore set of *nsems* OSS semaphores are created when either of the following is true:

- The value of **IPC_PRIVATE** is used for the *key* parameter.
- The *key* parameter does not already have a semaphore set ID associated with it, and (*semflg* **& IPC_CREAT**) is not 0 (zero).

After creating a new semaphore set ID, the **semget( )** function initializes the **semid_ds** structure associated with the ID as follows:

- The **sem_perm.cuid** and **sem_perm.uid** fields are set equal to the effective user ID of the calling process.
- The **sem_perm.cgid** and **sem_perm.gid** fields are set equal to the effective group ID of the calling process.
- The low-order nine bits of the **sem_perm.mode** field are set equal to the low-order nine bits of the *semflg* parameter.

- The **sem_nsems** field is set to the value of the *nsems* parameter.
- The **sem_otime** field is set to 0 (zero), and the **sem_ctime** field is set equal to the current time.

The **semget( )** function does not initialize the **sem** structure associated with each semaphore in the set. The individual OSS semaphores are initialized by using the **semctl( )** function with the **SETVAL** or **SETALL** value for the *cmd* parameter.

### Key Creation

The key represents a user-designated name for a given semaphore set. Keys are usually selected by calling the **ftok( )** function before calling the **semget( )** function. The **ftok( )** function returns a key based on a path and an interprocess communications identifier. This key is then passed to the **semget( )** function, which returns a semaphore set ID. The semaphore set ID is then used in calls to the **semop( )** and **semctl( )** functions.

### Propagation During Process Creation

Semaphore set IDs attached to a parent process are also attached to a child process. A semaphore set cannot be shared when a child process is created in a different processor than that used by the parent. If a process attempts to create a child process in a different processor while the parent process has any adjust-on-exit (**semadj**) value, the process creation fails and **errno** is set to [EHLDSEM].

### Cleaning Up Semaphores

An OSS semaphore remains allocated until it is removed. Normally, the **semctl( )** function is used with the **IPC_RMID** value of the *cmd* parameter to remove unneeded OSS semaphores.

The HP implementation of OSS environment semaphores does not provide facilities to detect or avoid deadlocks.

An allocated OSS semaphore set ID is not removed automatically when the last process using it terminates. Instead, the OSS semaphore set ID becomes inactive. The user must remove inactive OSS semaphore set IDs to avoid wasting system resources.

The status of OSS semaphore set IDs can be checked with the **ipcs** command. Inactive OSS semaphore set IDs can be removed with the **ipcrm** command.

### Semaphore Use Between Environments

OSS and Guardian environment nonprivileged binary semaphores coexist but do not interoperate.

Guardian environment processes cannot use OSS environment function calls for access to OSS semaphores. OSS environment processes can create and operate on nonprivileged binary semaphores through Guardian environment procedure calls.

### Use From the Guardian Environment

If called from a Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

## RETURN VALUES

Upon successful completion, a nonnegative semaphore set ID is returned. Otherwise, the **semget( )** function returns the value -1 and sets **errno** to indicate the condition.

## ERRORS

If any of the following conditions occurs, the **semget( )** function sets **errno** to the corresponding value:

[EACCES] A semaphore set ID already exists for the *key* parameter, but operation permission as specified by the low-order nine bits of the *semflg* parameter was not granted.

[EEXIST] A semaphore set ID already exists for the *key* parameter, but ((*semflg* **& IPC_CREAT**) **&& (***semflg* **& IPC_EXCL**)) is not equal to 0 (zero).

[EINVAL] One of the following conditions is true:

- A semaphore set ID already exists for the *key* parameter, but the number of semaphores in the set is less than *nsems* and *nsems* is not equal to 0 (zero).
- A semaphore set ID does not already exist, but the value of the *nsems* parameter is either less than or equal to 0 (zero) or greater than the system-defined limit.

[ENOENT] A semaphore set ID does not exist for the *key* parameter, and (*semflg* **& IPC_CREAT**) is equal to 0 (zero).

[ENOSPC] An attempt to create a new semaphore set ID exceeded the processor limit on the number of allowed semaphores.

[ENOTOSS] The calling process is not an OSS process. The requested operation is not supported from the Guardian environment.

## RELATED INFORMATION

Commands: **ipcrm(1)**, **ipcs(1)**.

Functions: **exec(2)**, **_exit(2)**, **fork(2)**, **ftok(3)**, **semctl(2)**, **semop(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** value [ENOTOSS] can be returned.

## NAME

**semop -** Performs semaphore operations

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

## SYNOPSIS

```
#include <sys/sem.h>

int semop(
        int semid,
        struct sembuf *sops,
        size_t nsops);
```

## PARAMETERS

*semid* Specifies the ID of the semaphore set.

*sops* Points to the user-defined array of **sembuf** structures that contain the semaphore operations.

*nsops* Specifies the number of **sembuf** structures in the array.

## DESCRIPTION

The **semop( )** function atomically performs a set of operations on the semaphores specified by the **sem_num** fields in the structures pointed to by the *sops* parameter and by the semaphore set ID specified as the *semid* parameter.

If a process cannot execute a specified operation on a single semaphore within the specified semaphore set, it cannot execute any operation on any semaphore within that set. Values related to any semaphores in the set remain unchanged by the failed call to the **semop( )** function. (The calling process's adjust-on-exit value, **semadj**, for the semaphore is also unaffected by a failed call. Refer to the **exit(3)** reference page for more information about **semadj** use.)

All processes waiting (suspended) for a semaphore are awakened when an operation occurs that could cause any one of them to proceed.

The semaphore operations are defined in the array pointed to by the *sops* parameter. The *sops* array contains *nsops* elements, each of which is represented by a **sembuf** structure.

The **sembuf** structure (from the **sys/sem.h** header file) is defined as follows:

```
struct sembuf {
        unsigned short int sem_num;
        short int          sem_op;
        short int          sem_flg;
};
```

The fields in the **sembuf** structure are defined as follows:

**sem_num** Specifies an individual semaphore within the semaphore set.

**sem_op** Specifies the operation to perform on the semaphore. The **sem_op** operation is specified as a negative integer, a positive integer, or 0 (zero). The effects of these operations are described later in this reference page.

**sem_flg** Specifies various flags for the operations. The possible values are as follows:

**SEM_UNDO** Instructs the system to adjust the process's adjust-on-exit value (**semadj**) for a modified semaphore. When the process exits, the system uses this value to restore the semaphore to the value it had before any modifications by the process. This flag is used to prevent locking of resources allocated through a semaphore by a process that no longer exists.

**IPC_NOWAIT**
Instructs the system to return an error condition if a requested operation would cause the process to sleep. If the system returns an error condition, none of the requested semaphore operations are performed.

If the **sem_op** field of the **sembuf** structure contains a negative integer and the calling process has alter access permission, the **semop( )** function does one of the following:

- If the semaphore's current value (in the **semval** field of the **sem** structure) is equal to or greater than the absolute value of **sem_op**, the absolute value of **sem_op** is subtracted from **semval**. If (**sem_flg** & **SEM_UNDO**) is not zero, the absolute value of **sem_op** is added to the calling process's **semadj** value for the semaphore.
- If the semaphore's current value (in the **semval** field of the **sem** structure) is less than the absolute value of **sem_op** and (**sem_flg** & **IPC_NOWAIT**) is not zero, **semop( )** returns immediately with an error.
- If the semaphore's current value (in the **semval** field of the **sem** structure) is less than the absolute value of **sem_op** and (**sem_flg** & **IPC_NOWAIT**) is 0 (zero), **semop( )** increments the semaphore's **semncnt** value (in the **sem** structure) and suspends the calling process. If the process is suspended, it sleeps until one of the following occurs:
  - The **semval** value becomes equal to or greater than the absolute value of **sem_op**. When this happens, the semaphore's **semncnt** value is decremented, the absolute value of **sem_op** is subtracted from **semval**, and, if (**sem_flg** & **SEM_UNDO**) is not zero, the absolute value of **sem_op** is added to the calling process's **semadj** value for the semaphore.
  - The semaphore set ID specified by the *semid* parameter is removed from the system. When this happens, **semop( )** returns immediately with an error.
  - The calling process catches a signal. When this happens, the semaphore's **semncnt** value is decremented and the calling process resumes execution as directed by the **sigaction( )** function.

If the **sem_op** field of the **sembuf** structure contains a positive integer and the calling process has alter access permission, **semop( )** adds the **sem_op** value to the semaphore's current **semval** value (in the **sem** structure). If (**sem_flg** & **SEM_UNDO**) is not zero, the **sem_op** value is subtracted from the calling process's **semadj** value for the semaphore.

If the **sem_op** field of the **sembuf** structure contains 0 (zero) and the calling process has read access permission, **semop( )** does one of the following:

- If the **semval** field of the **sem** structure contains 0 (zero), **semop( )** returns immediately.
- If **semval** is not zero and (**sem_flg** & **IPC_NOWAIT**) is not zero, **semop( )** returns immediately.
- If **semval** is not zero and (**sem_flg** & **IPC_NOWAIT**) is 0 (zero), **semop( )** increments the semaphore's **semzcnt** value (in the **sem** structure) and suspends the calling process. If the process is suspended, it sleeps until one of the following occurs:
  - — The **semval** value becomes 0 (zero). When this happens, the semaphore's **semzcnt** value (in the **sem** structure) is decremented.
  - — The semaphore set ID specified by the *semid* parameter is removed from the system. When this happens, **semop( )** returns immediately with an error.
  - — The calling process catches a signal. When this happens, the semaphore's **semzcnt** value is decremented and the calling process resumes execution as directed by the **sigaction( )** function.

The value of the **sempid** field in the **sem** structure for each OSS semaphore that is operated upon is set to the OSS process ID of the calling process.

### Use From the Guardian Environment

If called from a Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

## RETURN VALUES

Upon successful completion, the **semop( )** function returns the value 0 (zero).

If the **semop( )** function fails, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **semop( )** function sets **errno** to the corresponding value:

| | |
|---|---|
| [E2BIG] | The value of the *nsops* parameter is greater than the system-defined maximum. |
| [EACCES] | The calling process does not have the required access permission. |
| [EAGAIN] | The value of (**sem_flg** && **IPC_NOWAIT**) is **TRUE**, but the requested operation would cause the calling process to be suspended. |
| [EFAULT] | The address used for the *sops* parameter is invalid. |
| [EFBIG] | The **sem_num** field of the **sembuf** structure is less than 0 (zero) or greater than or equal to the number of semaphores in the set identified by the *semid* parameter. |
| [EIDRM] | The semaphore set ID specified by the *semid* parameter was removed from the system while the process was waiting for it. |
| [EINTR] | The **semop( )** function was interrupted by a signal. |

[EINVAL] One of the following conditions is true:

- The *semid* parameter is not a valid semaphore set ID.
- The number of semaphores for which the **SEM_UNDO** flag is specified exceeds the system-defined limit.

[ENOSPC] One of the following conditions is true:

- The system-defined limit on the number of undo entries for an undo structure would be exceeded.
- The system-defined limit on the number of **SEM_UNDO** structures for a single processor would be exceeded.
- The number of **semadj** values for the processor would exceed the system limit.

[ENOTOSS] The calling process is not an OSS process. The requested operation is not supported from the Guardian environment.

[ERANGE] On of the following conditions exists:

- An operation caused a **semval** value in a **sem** structure to overflow the system-defined limit.
- An operation caused an adjust-on-exit (**semadj**) value to exceed the system-defined limit.

## RELATED INFORMATION

Functions: **exec(2)**, **_exit(2)**, **fork(2)**, **semctl(2)**, **semget(2)**, **sigaction(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT] and [ENOTOSS] can be returned.

## NAME

**send -** Sends a message on a connected socket

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t send(
        int socket,
        const void *buffer,
        size_t length,
        int flags);
```

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*buffer* Points to the buffer containing the message to send.

*length* Specifies the length in bytes of the message to send.

*flags* Is a value that controls message transmission. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_DONTROUTE**
Sends without using routing tables. (Not recommended, use for debugging only.)

**MSG_OOB** Sends out-of-band data on sockets that support out-of-band communications.

## DESCRIPTION

The **send( )** function begins transmission of a message to a peer socket. The **send( )** function sends a message only when the socket is connected.

The length of the message to be sent is specified by the *length* parameter. If the message is too long to pass through the underlying protocol, the **send( )** function fails and does not transmit the message.

Successful completion of a call to **send( )** does not imply successful delivery of the message. A return value of -1 indicates only locally detected errors.

If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **send( )** function blocks until space is available. If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **send( )** function fails and sets **errno** to [EWOULDBLOCK].

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **send( )** or **send64_( )** may be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **send64_( )** must be called.

32-bit Guardian and 64-bit OSS clients can pass 32-bit pointers and 64-bit pointers to **send64_( )**.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When data can be sent, a call to the **select( )** function indicates that the file descriptor for the socket is ready for writing.

Calling the **send( )** function with a *flags* parameter of 0 (zero) is identical to calling the **write( )** function.

To use the **send( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_sendx(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

A call to the thread-aware **send( )** function with a *flags* parameter value of 0 (zero) is identical to a call to the thread-aware **write( )** function.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the **send( )** function returns the number of bytes sent. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **send( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

This error is also returned if the **send( )** function is thread-aware and the socket becomes invalid (is closed by another thread).

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EDESTADDRREQ]
The socket is not connection-oriented and no peer address is set.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EINTR] A signal interrupted the function before any data was transmitted.

This error is also returned if the **send( )** function is thread-aware and a signal received from the **pthread_kill( )** function is not blocked, ignored, or handled.

[EIO] An input or output error occurred.

[EMSGSIZE] The message is too large to be sent all at once, as required by the socket.

[ENETDOWN]
The local interface used to reach the destination is down.

[ENETUNREACH]
No route to the network or host is present.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] The socket either is not connected or has not had the peer socket previously specified.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[EPIPE] One of the following conditions occurred:

- An attempt was made to send a message on a socket that is shut down for writing.
- An attempt was made to send a message on a connection-oriented socket and the peer socket is closed or shut down for reading. The **SIGPIPE** signal is also sent to the calling process.

[EWOULDBLOCK]
The socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the operation would block.

**RELATED INFORMATION**

Functions: **connect(2)**, **fcntl(2)**, **getsockopt(2)**, **recv(2)**, **recvfrom(2)**, **recvmsg(2)**, **select(2)**, **sendmsg(2)**, **sendto(2)**, **setsockopt(2)**, **sockatmark(2)**, **shutdown(2)**, **socket(2)**, **spt_sendx(2)**.

**STANDARDS CONFORMANCE**

The HP implementation does not return the **errno** value [ENOSR].

The following are HP extensions to the XPG4 specification:

- The **errno** value [ECONNRESET] can be returned when the transport-provider process is not available.
- For systems running J06.07 and later J-series RVUs or H06.18 and later H-series RVUs, the **errno** value [ENOMEM] can be returned when there is not enough system memory available to complete the operation.

This function is an extension to the XPG4 Version 2 specification.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**send64_** - Sends a message on a connected socket

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

long long send64_(
        int socket,
        const void _ptr64 *buffer,
        unsigned long long length,
        int flags);
```

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*buffer* Points to the buffer containing the message to send.

*length* Specifies the length in bytes of the message to send.

*flags* Is a value that controls message transmission. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_DONTROUTE**
Sends without using routing tables. (Not recommended, use for debugging only.)

**MSG_OOB** Sends out-of-band data on sockets that support out-of-band communications.

## DESCRIPTION

The **send64_( )** function begins transmission of a message to a peer socket. The **send64_( )** function sends a message only when the socket is connected.

The length of the message to be sent is specified by the *length* parameter. If the message is too long to pass through the underlying protocol, the **send64_( )** function fails and does not transmit the message.

Successful completion of a call to **send64_( )** does not imply successful delivery of the message. A return value of -1 indicates only locally detected errors.

If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **send64_( )** function blocks until space is available. If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **send64_( )** function fails and sets **errno** to [EWOULDBLOCK].

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **send( )** or **send64_( )** may be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **send64_( )** must be called.

32-bit Guardian and 64-bit OSS clients can pass 32-bit pointers and 64-bit pointers to **send64_( )**.

**NOTES**

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When data can be sent, a call to the **select( )** function indicates that the file descriptor for the socket is ready for writing.

Calling the **send64_( )** function with a *flags* parameter of 0 (zero) is identical to calling the **write64_( )** function.

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **send64_( )** function returns the number of bytes sent. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **send64_( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EDESTADDRREQ]
The socket is not connection-oriented and no peer address is set.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EINTR] A signal interrupted the function before any data was transmitted.

[EIO] An input or output error occurred.

[EMSGSIZE] The message is too large to be sent all at once, as required by the socket.

[ENETDOWN]
The local interface used to reach the destination is down.

[ENETUNREACH]
No route to the network or host is present.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] The socket either is not connected or has not had the peer socket previously specified.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[EPIPE] One of the following conditions occurred:

- An attempt was made to send a message on a socket that is shut down for writing.
- An attempt was made to send a message on a connection-oriented socket and the peer socket is closed or shut down for reading. The **SIGPIPE** signal is also sent to the calling process.

[EWOULDBLOCK]
The socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the operation would block.

### RELATED INFORMATION

Functions: **connect(2)**, **fcntl(2)**, **getsockopt(2)**, **recv(2)**, **recv64_(2)**, **recvfrom(2)**, **recvfrom64_(2)**, **recvmsg(2)**, **recvmsg64_(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **sendmsg64_(2)**, **sendto(2)**, **sendto64_(2)**, **setsockopt(2)**, **sockatmark(2)**, **shutdown(2)**, **socket(2)**.

### STANDARDS CONFORMANCE

This API is an HP extension and is not standards conformant.

## NAME

**sendmsg -** Sends a message on a socket using a message structure

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library: **/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t sendmsg(
        int socket,
        const struct msghdr *message,
        int flags);
```

## PARAMETERS

*socket*
: Specifies the file descriptor of the socket.

*message*
: Points to a **msghdr** structure containing both the destination address for the outgoing message and the buffers for the outgoing message. The length and format of the address depend on the address family for the socket. The **msg_flags** member of the structure is ignored. For:

  **AF_INET** sockets
  : A pointer in **msghdr** to the address structure **sockaddr_in** must be cast as a **struct sockaddr**.

  **AF_INET6** sockets
  : A pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**.

  **AF_UNIX** sockets
  : A pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*flags*
: Is a value that controls message transmission. The value of the *flags* parameter is formed by bitwise ORing zero or more of these values:

  **MSG_DONTROUTE**
  : Sends without using routing tables. (Not recommended; use only for debugging purposes.)

  **MSG_OOB**
  : Sends out-of-band data on sockets that support out-of-band communications.

## DESCRIPTION

The **sendmsg( )** function sends a message through a connection-oriented or connectionless socket. If the socket is connectionless, the message is sent to the address specified in the **msghdr** structure. If the socket is connection-oriented, the destination address in the **msghdr** structure is ignored.

Successful completion of a call to **sendmsg( )** does not imply successful delivery of the message. A return value of -1 indicates only locally detected errors.

If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **sendmsg( )** function blocks until space is available. If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **sendmsg( )** function fails and sets **errno** to [EWOULDBLOCK].

In the **msghdr** structure, the **msg_control** and **msg_controllen** members specify the ancillary data buffer that can be used only by sockets in the **AF_UNIX** domain to pass file descriptors to another process on the same node. The **msg_control** member can be a null pointer if ancillary data is not desired or required. If the **msg_control** member is nonnull, it points to an ancillary data buffer consisting of a **cmsghdr** structure followed by one to sixteen file descriptors. The **msg_controllen** member specifies the size of the ancillary data buffer.

If **sendmsg( )** is called with an ancillary data buffer, the members of the **cmsghdr** structure must be set as follows:

- The **cmsg_level** member must be set to **SOL_SOCKET**.
- The **cmsg_type** member must be set to **SCM_RIGHTS**.
- The value of the **cmsg_len** member must be equal to the value of the **msg_controllen** member of the **msghdr** structure.

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **sendmsg( )** must be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **sendmsg64_( )** must be called.

To pass a 32-bit pointer from a 64-bit OSS client, **sendmsg( )** must be called.

To pass a 64-bit pointer from a 64-bit OSS client, **sendmsg_( )** must be called.

**NOTES**

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When data can be sent, a call to the **select( )** function indicates that the file descriptor for the socket is ready for writing.

For systems running **AF_UNIX** Release 2 software:

- Sockets created in compatibility mode can communicate with each other but cannot communicate with sockets in portability mode.
- Sockets created in portability mode can communicate with each other but cannot communicate with sockets created in compatibility mode.

For J06.07 and later J-series RVUs and H06.18 and later H-series RVUs, if a memory resource allocation error occurs while attempting this operation, the operation succeeds but the resulting file descriptor is not usable. All subsequent file operations that attempt to use the file descriptor fail with the error [EBADF].

To use the **sendmsg( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_sendmsgx(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **sendmsg( )** function returns the number of normal bytes sent. Ancillary data, if present, is not counted in the total number of bytes sent.

If the **sendmsg( )** function call fails, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **sendmsg( )** function sets **errno** to the corresponding value:

[EACCES] The socket is in the **AF_UNIX** domain and either search permission is denied for a component of the pathname in the **msghdr** structure or write access to the specified socket is denied.

[EAFNOSUPPORT]
Addresses in the specified address family cannot be used with this socket.

[EBADF] One of these conditions exists:

- The *socket* parameter is not a valid file descriptor.
- The socket is in the **AF_UNIX** domain, and one or more of the file descriptors being passed is invalid.

This error is also returned if the **sendmsg( )** function is thread-aware and the socket becomes invalid (is closed by another thread).

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EDESTADDRREQ]
The socket is not connection-oriented, no peer address is set, and no destination address is specified.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EINTR] A signal interrupted the function before any data was transmitted.

This error is also returned if the **sendmsg( )** function is thread-aware and a signal received from the **pthread_kill( )** function is not blocked, ignored, or handled.

[EINVAL] One of these conditions occurred:

- The socket is in the **AF_UNIX** domain, and the **msg_control** member contains either more than 16 file descriptors or fewer than 1 file descriptor.
- The socket is in the **AF_UNIX** domain, and an attempt was made to send more than one **cmsghdr** structure.
- The socket is in the **AF_UNIX** domain, and the value of the **cmsg_len** member is not equal to the value of the **msg_controllen** member.
- The socket is in the **AF_UNIX** domain, and the **cmsg_type** member is not equal to **SCM_RIGHTS**.
- The sum of the values specified for the **msg_iovlen** member of the **msghdr** structure is too large for a data item of type **ssize_t**.

[EIO] The socket is in the **AF_UNIX** domain, and the transport agent failed to inherit the file descriptors being passed, or an input or output error occurred.

[ELOOP] The socket is in the **AF_UNIX** domain, and too many symbolic links were encountered in translating the pathname specified by the **msghdr** structure.

[EMSGSIZE] The message is too large to be sent all at once, as required by the socket.

[ENAMETOOLONG]
The socket is in the **AF_UNIX** domain, and one of these conditions exists:

- The pathname in the **msghdr** structure exceeds **PATH_MAX** characters.
- A component of the pathname in the **msghdr** structure exceeds **NAME_MAX** characters.
- The intermediate result of pathname resolution when a symbolic link is part of the pathname in the **msghdr** structure exceeds **PATH_MAX** characters.

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOENT] The socket is in the **AF_UNIX** domain, and one of these conditions occurred:

- A component of the pathname in the **msghdr** structure does not name an existing file.

- The **msghdr** structure specifies an empty string as a pathname.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOPROTOOPT]
The socket is in the **AF_UNIX** domain, and the **cmsg_level** member is not equal to **SOL_SOCKET**.

[ENOTCONN] The socket is connection-oriented but is not connected.

[ENOTDIR] The socket is in the **AF_UNIX** domain, and the pathname specified by the **msghdr** structure contains a component that is not a directory.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[EPERM] The address included in the *message* parameter is bound to a socket whose mode is different than the mode of the socket specified by the *socket* parameter.

[EPIPE] One of these conditions occurred:

- An attempt was made to send a message on a socket that is shut down for writing.
- An attempt was made to send a message on a connection-oriented socket, and the peer socket is closed or shut down for reading. The **SIGPIPE** signal is also sent to the calling process.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set), and the operation would block.

## RELATED INFORMATION

Functions: **fcntl(2)**, **getsockopt(2)**, **recv(2)**, **recvfrom(2)**, **recvmsg(2)**, **select(2)**, **send(2)**, **sendto(2)**, **setsockopt(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **socketpair(2)**, **spt_sendmsgx(2)**.

## STANDARDS CONFORMANCE

The HP implementation does not return the **errno** value [ENOSR].

HP extensions to the XPG4 specification are:

- The **errno** value [ECONNRESET] can be returned when the transport-provider process is not available.
- The **errno** value [ENOPROTOOPT] can be returned.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**sendmsg64_ -** Sends a message on a socket using a message structure

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

long long sendmsg64_(
        int socket,
        const struct msghdr64 _ptr64 *message,
        int flags);
```

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*message* Points to a **msghdr64** structure containing both the destination address for the outgoing message and the buffers for the outgoing message. The length and format of the address depend on the address family for the socket. The **msg_flags** member of the structure is ignored. For:

**AF_INET** sockets
A pointer in **msghdr64** to the address structure **sockaddr_in** must be cast as a **struct sockaddr**.

**AF_INET6** sockets
A pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**.

**AF_UNIX** sockets
A pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*flags* Is a value that controls message transmission. The value of the *flags* parameter is formed by bitwise ORing zero or more of these values:

**MSG_DONTROUTE**
Sends without using routing tables. (Not recommended; use only for debugging purposes.)

**MSG_OOB** Sends out-of-band data on sockets that support out-of-band communications.

## DESCRIPTION

The **sendmsg64_( )** function sends a message through a connection-oriented or connectionless socket. If the socket is connectionless, the message is sent to the address specified in the **msghdr64** structure. If the socket is connection-oriented, the destination address in the **msghdr64** structure is ignored.

Successful completion of a call to **sendmsg64_( )** does not imply successful delivery of the message. A return value of -1 indicates only locally detected errors.

If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **sendmsg64_( )** function blocks until space is available. If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **sendmsg64_( )** function fails and sets **errno** to [EWOULDBLOCK].

In the **msghdr64** structure, the **msg_control** and **msg_controllen** members specify the ancillary data buffer that can be used only by sockets in the **AF_UNIX** domain to pass file descriptors to another process on the same node. The **msg_control** member can be a null pointer if ancillary data is not desired or required. If the **msg_control** member is nonnull, it points to an ancillary data buffer consisting of a **cmsghdr** structure followed by one to sixteen file descriptors. The **msg_controllen** member specifies the size of the ancillary data buffer.

If **sendmsg64_( )** is called with an ancillary data buffer, the members of the **cmsghdr** structure must be set as follows:

- The **cmsg_level** member must be set to **SOL_SOCKET**.
- The **cmsg_type** member must be set to **SCM_RIGHTS**.
- The value of the **cmsg_len** member must be equal to the value of the **msg_controllen** member of the **msghdr64** structure.

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **sendmsg( )** must be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **sendmsg64_( )** must be called.

To pass a 32-bit pointer from a 64-bit OSS client, **sendmsg( )** must be called.

To pass a 64-bit pointer from a 64-bit OSS client, **sendmsg( )** must be called.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When data can be sent, a call to the **select( )** function indicates that the file descriptor for the socket is ready for writing.

For **AF_UNIX** sockets:

- Sockets created in compatibility mode can communicate with each other but cannot communicate with sockets in portability mode.
- Sockets created in portability mode can communicate with each other but cannot communicate with sockets created in compatibility mode.

If a memory resource allocation error occurs while attempting this operation, the operation succeeds but the resulting file descriptor is not usable. All subsequent file operations that attempt to use the file descriptor fail with the error [EBADF].

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the **sendmsg64_( )** function returns the number of normal bytes sent. Ancillary data, if present, is not counted in the total number of bytes sent.

If the **sendmsg64_( )** function call fails, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **sendmsg64_( )** function sets **errno** to the corresponding value:

[EACCES] The socket is in the **AF_UNIX** domain and either search permission is denied for a component of the pathname in the **msghdr64** structure or write access to the specified socket is denied.

[EAFNOSUPPORT] Addresses in the specified address family cannot be used with this socket.

[EBADF] One of these conditions exists:

- The *socket* parameter is not a valid file descriptor.
- The socket is in the **AF_UNIX** domain, and one or more of the file descriptors being passed is invalid.

[ECONNRESET] One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EDESTADDRREQ] The socket is not connection-oriented, no peer address is set, and no destination address is specified.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EINTR] A signal interrupted the function before any data was transmitted.

[EINVAL] One of these conditions occurred:

- The socket is in the **AF_UNIX** domain, and the **msg_control** member contains either more than 16 file descriptors or fewer than 1 file descriptor.
- The socket is in the **AF_UNIX** domain, and an attempt was made to send more than one **cmsghdr** structure.
- The socket is in the **AF_UNIX** domain, and the value of the **cmsg_len** member is not equal to the value of the **msg_controllen** member.
- The socket is in the **AF_UNIX** domain, and the **cmsg_type** member is not equal to **SCM_RIGHTS**.
- The sum of the values specified for the **msg_iovlen** member of the **msghdr64** structure is too large for a long long data item.

[EIO] The socket is in the **AF_UNIX** domain, and the transport agent failed to inherit the file descriptors being passed, or an input or output error occurred.

[ELOOP] The socket is in the **AF_UNIX** domain, and too many symbolic links were encountered in translating the pathname specified by the **msghdr64** structure.

[EMSGSIZE] The message is too large to be sent all at once, as required by the socket.

[ENAMETOOLONG]
The socket is in the **AF_UNIX** domain, and one of these conditions exists:

- The pathname in the **msghdr64** structure exceeds **PATH_MAX** characters.
- A component of the pathname in the **msghdr64** structure exceeds **NAME_MAX** characters.
- The intermediate result of pathname resolution when a symbolic link is part of the pathname in the **msghdr64** structure exceeds **PATH_MAX** characters.

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOENT] The socket is in the **AF_UNIX** domain, and one of these conditions occurred:

- A component of the pathname in the **msghdr64** structure does not name an existing file.
- The **msghdr64** structure specifies an empty string as a pathname.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOPROTOOPT]
The socket is in the **AF_UNIX** domain, and the **cmsg_level** member is not equal to **SOL_SOCKET**.

[ENOTCONN] The socket is connection-oriented but is not connected.

[ENOTDIR] The socket is in the **AF_UNIX** domain, and the pathname specified by the **msghdr64** structure contains a component that is not a directory.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[EPERM] The address included in the *message* parameter is bound to a socket whose mode is different than the mode of the socket specified by the *socket* parameter.

[EPIPE] One of these conditions occurred:

- An attempt was made to send a message on a socket that is shut down for writing.

- An attempt was made to send a message on a connection-oriented socket, and the peer socket is closed or shut down for reading. The **SIGPIPE** signal is also sent to the calling process.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set), and the operation would block.

**RELATED INFORMATION**

Functions: **fcntl(2)**, **getsockopt(2)**, **recv(2)**, **recv64_(2)**, **recvfrom(2)**, **recvfrom64_(2)**, **recvmsg(2)**, **recvmsg64_(2)**, **select(2)**, **send(2)**, **send64_(2)**, **sendmsg(2)**, **sendto(2)**, **sendto64_(2)**, **setsockopt(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **socketpair(2)**.

**STANDARDS CONFORMANCE**

This API is an HP extension and is not standards conformant.

## NAME

**sendto -** Sends a message on a socket

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t sendto(
        int socket,
        const void *message,
        size_t length,
        int flags,
        const struct sockaddr *dest_addr,
        socklen_t dest_len);
```

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*message* Points to the buffer containing the message to be sent.

*length* Specifies the length in bytes of the message to be sent.

*flags* Is a value that controls message transmission. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_DONTROUTE**
Sends without using routing tables. (Not recommended; use for debugging purposes only.)

**MSG_OOB** Sends out-of-band data on sockets that support out-of-band communications.

*dest_addr* Points to a **sockaddr** structure that contains the destination address. The length and format of the address depends on the address family of the socket. For:

**AF_INET** sockets
A pointer to the address structure **sockaddr_in** must be cast as a **struct sockaddr**.

**AF_INET6** sockets
A pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**.

**AF_UNIX** sockets
A pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*dest_len* Specifies the length of the **sockaddr** structure pointed to by the *dest_addr* parameter.

**DESCRIPTION**

The **sendto( )** function sends a message through a connection-oriented or connectionless socket. If the socket is connectionless, the message is sent to the address specified in the **sockaddr** structure pointed to by the *dest_addr* parameter. If the socket is connection-oriented, the *dest_addr* parameter is ignored.

Successful completion of a call to **sendto( )** does not imply successful delivery of the message. A return value of -1 indicates only locally detected errors.

If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **sendto( )** function blocks until space is available. If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **sendto( )** function fails and sets **errno** to [EWOULDBLOCK].

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **sendto( )** or **sendto64_( )** may be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **sendto64_( )** must be called.

32-bit Guardian and 64-bit OSS clients can pass 32-bit pointers and 64-bit pointers to **sendto64_( )**.

**NOTES**

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When data can be sent, a call to the **select( )** function indicates that the file descriptor for the socket is ready for writing.

For systems running **AF_UNIX** Release 2 software:

- Sockets created in compatibility mode can communicate with each other but cannot communicate with sockets in portability mode.
- Sockets created in portability mode can communicate with each other but cannot communicate with sockets created in compatibility mode.

To use the **sendto( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_sendtox(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **sendto( )** function returns the number of bytes sent. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **sendto( )** function sets **errno** to the corresponding value:

[EACCES] The socket is in the **AF_UNIX** domain and either search permission is denied for a component of the pathname in the **sockaddr** structure, or write access to the specified socket is denied.

[EAFNOSUPPORT]
Addresses in the specified address family cannot be used with this socket.

[EBADF] The *socket* parameter is not a valid file descriptor.

This error is also returned if the **sendto( )** function is thread-aware and the socket becomes invalid (is closed by another thread).

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EDESTADDRREQ]
The socket is not connection-oriented and does not have its peer address set, and no destination address was specified.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EHOSTUNREACH]
The destination host cannot be reached.

[EINTR] A signal interrupted the function before any data was transmitted.

This error is also returned if the **sendto( )** function is thread-aware and a signal received from the **pthread_kill( )** function is not blocked, ignored, or handled.

[EIO] The socket is in the **AF_UNIX** domain and an input or output error occurred.

[EINVAL] The *dest_len* parameter is not a valid length for the address family.

[ELOOP] The socket is in the **AF_UNIX** domain and too many symbolic links were encountered in translating the pathname in the **sockaddr** structure.

[EMSGSIZE] The message is too large to be sent all at once, as required by the socket.

[ENAMETOOLONG]
The socket is in the **AF_UNIX** domain and one of the following conditions exists:

- The pathname in the **sockaddr** structure exceeds **PATH_MAX** characters.
- A component of the pathname in the **sockaddr** structure exceeds **NAME_MAX** characters.
- The intermediate result of pathname resolution when a symbolic link is part of the pathname in the **sockaddr** structure exceeds **PATH_MAX** characters.

The **pathconf( )** function can be called to obtain the applicable limits.

[ENETDOWN]
The local interface used to reach the destination is down.

[ENETUNREACH]
No route to the network or host is present.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time might succeed.

[ENOENT] The socket is in the **AF_UNIX** domain and one of the following conditions exists:

- A component of the pathname specified in the **sockaddr** structure does not name an existing file.
- The **sockaddr** structure specifies an empty string as a pathname.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] The socket is connection-oriented but is not connected.

[ENOTDIR] The socket is in the **AF_UNIX** domain and the pathname in the **sockaddr** structure contains a component that is not a directory.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[EPERM] The file name specified by the *dest_addr* parameter is bound to a socket whose mode is different than the mode of the socket specified by the *socket* parameter.

[EPIPE] One of the following conditions occurred:

- An attempt was made to send a message on a socket that is shut down for writing.
- An attempt was made to send a message on a connection-oriented socket and the peer socket is closed or shut down for reading. The **SIGPIPE** signal is also sent to the calling process.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the operation would block.

## RELATED INFORMATION

Functions: **fcntl(2)**, **getsockopt(2)**, **recv(2)**, **recvfrom(2)**, **recvmsg(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **setsockopt(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **spt_sendtox(2)**.

## STANDARDS CONFORMANCE

The HP implementation does not return the **errno** values [EISCONN] or [ENOSR].

The following are HP extensions to the XPG4 specification:

- The **errno** value [ECONNRESET] can be returned when the transport-provider process is not available.

This function is an extension to the XPG4 Version 2 specification.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**sendto64_ -** Sends a message on a socket

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <sys/socket.h>**

**long long sendto64_(**
  **int** *socket***,**
  **const void _ptr64 ****message***,**
  **unsigned long long** *length***,**
  **int** *flags***,**
  **const struct sockaddr _ptr64 ****dest_addr***,**
  **socklen_t** *dest_len***);**

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*message* Points to the buffer containing the message to be sent.

*length* Specifies the length in bytes of the message to be sent.

*flags* Is a value that controls message transmission. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_DONTROUTE**
Sends without using routing tables. (Not recommended; use for debugging purposes only.)

**MSG_OOB** Sends out-of-band data on sockets that support out-of-band communications.

*dest_addr* Points to a **sockaddr** structure that contains the destination address. The length and format of the address depends on the address family of the socket. For:

**AF_INET** sockets
A pointer to the address structure **sockaddr_in** must be cast as a **struct sockaddr**.

**AF_INET6** sockets
A pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**.

**AF_UNIX** sockets
A pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*dest_len* Specifies the length of the **sockaddr** structure pointed to by the *dest_addr* parameter.

**DESCRIPTION**

The **sendto64_( )** function sends a message through a connection-oriented or connectionless socket. If the socket is connectionless, the message is sent to the address specified in the **sockaddr** structure pointed to by the *dest_addr* parameter. If the socket is connection-oriented, the *dest_addr* parameter is ignored.

Successful completion of a call to **sendto64_( )** does not imply successful delivery of the message. A return value of -1 indicates only locally detected errors.

If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **sendto64_( )** function blocks until space is available. If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **sendto64_( )** function fails and sets **errno** to [EWOULDBLOCK].

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **sendto( )** or **sendto64_( )** may be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **sendto64_( )** must be called.

32-bit Guardian and 64-bit OSS clients can pass 32-bit pointers and 64-bit pointers to **sendto64_( )**.

**NOTES**

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

When data can be sent, a call to the **select( )** function indicates that the file descriptor for the socket is ready for writing.

Sockets created in compatibility mode can communicate with each other but cannot communicate with sockets in portability mode.

Sockets created in portability mode can communicate with each other but cannot communicate with sockets created in compatibility mode.

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **sendto64_( )** function returns the number of bytes sent. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **sendto64_( )** function sets **errno** to the corresponding value:

[EACCES] The socket is in the **AF_UNIX** domain and either search permission is denied for a component of the pathname in the **sockaddr** structure, or write access to the specified socket is denied.

[EAFNOSUPPORT]
Addresses in the specified address family cannot be used with this socket.

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EDESTADDRREQ]
The socket is not connection-oriented and does not have its peer address set, and no destination address was specified.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EHOSTUNREACH]
The destination host cannot be reached.

[EINTR] A signal interrupted the function before any data was transmitted.

[EIO] The socket is in the **AF_UNIX** domain and an input or output error occurred.

[EINVAL] The *dest_len* parameter is not a valid length for the address family.

[ELOOP] The socket is in the **AF_UNIX** domain and too many symbolic links were encountered in translating the pathname in the **sockaddr** structure.

[EMSGSIZE] The message is too large to be sent all at once, as required by the socket.

[ENAMETOOLONG]
The socket is in the **AF_UNIX** domain and one of the following conditions exists:

- The pathname in the **sockaddr** structure exceeds **PATH_MAX** characters.
- A component of the pathname in the **sockaddr** structure exceeds **NAME_MAX** characters.
- The intermediate result of pathname resolution when a symbolic link is part of the pathname in the **sockaddr** structure exceeds **PATH_MAX** characters.

The **pathconf( )** function can be called to obtain the applicable limits.

[ENETDOWN]
The local interface used to reach the destination is down.

[ENETUNREACH]
No route to the network or host is present.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time might succeed.

[ENOENT] The socket is in the **AF_UNIX** domain and one of the following conditions exists:

- A component of the pathname specified in the **sockaddr** structure does not name an existing file.
- The **sockaddr** structure specifies an empty string as a pathname.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTCONN] The socket is connection-oriented but is not connected.

[ENOTDIR] The socket is in the **AF_UNIX** domain and the pathname in the **sockaddr** structure contains a component that is not a directory.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[EPERM] The file name specified by the *dest_addr* parameter is bound to a socket whose mode is different than the mode of the socket specified by the *socket* parameter.

[EPIPE] One of the following conditions occurred:

- An attempt was made to send a message on a socket that is shut down for writing.
- An attempt was made to send a message on a connection-oriented socket and the peer socket is closed or shut down for reading. The **SIGPIPE** signal is also sent to the calling process.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the operation would block.

## RELATED INFORMATION

Functions: **fcntl(2)**, **getsockopt(2)**, **recv(2)**, **recv64_(2)**, **recvfrom(2)**, **recvfrom64_(2)**, **recvmsg(2)**, **recvmsg64_(2)**, **select(2)**, **send(2)**, **send64_(2)**, **sendmsg(2)**, **sendmsg64_(2)**, **sendto(2)**, **setsockopt(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **spt_sendtox(2)**.

## STANDARDS CONFORMANCE

This API is an HP extension and is not standards conformant.

## NAME

**setegid** - Sets the effective group ID of the calling process

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsecsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zsecdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/ysecdll**

## SYNOPSIS

**#include <sys/types.h>** /* optional except for POSIX.1 */

**#include <unistd.h>**

**int setegid( gid_t** *egid* **);**

## PARAMETERS

*egid* Specifies the new effective group ID.

## DESCRIPTION

The **setegid()** function sets the effective group ID of the current process to the value specified by *egid* parameter.

The process that calls this function must have appropriate privileges. A process without appropriate privileges can set the effective group ID only if the *egid* parameter is equal to either the real or saved-set-group-ID of the process.

The value of the *egid* **parameter must be in the range 0 (zero) through 65535.**

The real group ID, the saved-set-group-ID, and the group list of the calling process are not changed.

## NOTES

This function is supported on systems running J06.07 and later J-series RVUs, H06.18 and later H-series RVUs, and G06.33 and later G-series RVUs.

This function does not set the default file security of a process. To set the default file security for a process, use the PROCESS_SETINFO_ Guardian procedure call with item code 41.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occur, the **setegid()** function sets **errno** to the corresponding value:

[EINVAL] The value of the *egid* parameter is invalid or out of range.

[EPERM] The current process does not have appropriate privileges and the *egid* parameter does not match the real group ID or the saved-set-group-ID of the process.

## RELATED INFORMATION

Functions: **getegid(2)**, **getgid(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- If both the real and effective group IDs are changed so that they differ from each other and from the saved-set-group-ID, then the saved-set-group-ID is set to the value of the effective group ID.

- A process without appropriate privileges can set the effective group ID if the new effective group ID matches a group ID in the group list of the process.

## NAME

**seteuid** - Sets the effective user ID of the calling process

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsecsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zsecdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/ysecdll**

## SYNOPSIS

**#include <sys/types.h> /* optional except for POSIX.1 */**

**#include <unistd.h>**

**int seteuid( uid_t** *euid* **);**

## PARAMETERS

*euid* Specifies the new effective user ID.

## DESCRIPTION

The **seteuid( )** function sets the effective user ID of the calling process to the value specified by the *euid* parameter.

The process that calls this function must have appropriate privileges. A process without appropriate privileges can set the effective user ID only if the *euid* parameter is equal to either the real or saved-set-user-ID of the process.

The value of the *euid* parameter must be in the range 0 (zero) through 65535.

The real user ID and the saved-set-user-ID of the calling process are not changed.

### Use on Guardian Objects

Changing the effective user ID sets the process access ID (PAID) to the value of the effective user ID.

## NOTES

This function is supported on systems running J06.07 and later J-series RVUs, H06.18 and later H-series RVUs, and G06.33 and later G-series RVUs.

This function does not set the default file security of a process. To set the default file security for a process, use the PROCESS_SETINFO_ Guardian procedure call with item code 41.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any these conditions occur, the **seteuid( )** function sets **errno** to the corresponding value:

[EINVAL] The value of the *euid* parameter is invalid or out of range.

[EPERM] The current process does not have appropriate privileges and the *euid* parameter does not match the real user ID or the saved-set-user-ID of the process.

## RELATED INFORMATION

Functions: **getuid(2)**, **setuid(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- If both the real and effective user IDs are changed so that they differ from each other and from the saved-set-user-ID, then the saved-set-user-ID is set to the value of the effective user ID.

## NAME

**setfilepriv -** Sets one or more file privileges for an executable file

## LIBRARY

H-series and J-series native Guardian Procesess: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/privileges.h>**

**int setfilepriv(**
**const char** ***path*, **
**const unsigned char** ***fileprivs***
**);**

## PARAMETERS

*path* Points to the OSS pathname of the executable file.

*fileprivs* Points to the bit pattern that determines the privileges for the file.

## DESCRIPTION

The **setfilepriv( )** function sets the file privileges of the OSS regular file or Guardian disk file specified in the *path* parameter according to the bit pattern specified by the *fileprivs* parameter.

File privileges are not supported for file types other than OSS regular files or Guardian disk files. File privileges are ignored for files that are not executable files, DLLs, or user libraries. For example, file privileges are ignored for shell scripts and TACL scirpts.

The *fileprivs* parameter is constructed by logically ORing one or more of these symbols, which are defined in the **sys/privileges.h** header file:

PRIVNONE Resets the file privileges so that file has no special privileges.

PRIVSETID If the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) runs an executable file that has this file privilege, the resultant process is permitted to perform a privileged switch (such as by using the **setuid( )** function) to another user ID, group ID, or both to access files in a restricted-access fileset.

PRIVSOARFOPEN
If a locally-authenticated member of the Safeguard SECURITY_OSS_ADMINISTRATOR (SOA) group runs an executable file that has this file privilege, the resultant process is permitted to perform additional system calls needed to back up and restore files in a restricted-access fileset. These system calls include **open( )**, **open64( )**, **creat( )**, **creat64( )**, **link( )**, **remove_oss( )**, **unlink( )**, **rmdir( )**, and **utime( )**,

## NOTES

This function is supported on systems running J06.11 or later J-series RVUs or H06.22 or later H-series RVUs only

Only Members of Safeguard SECURITY-PRV-ADMINISTRATOR (SEC-PRIV-ADMIN or SPA) group are permitted to explicitly set or reset file privileges. Therefore only members can set the PRIVSOARFOPEN file privilege on the Backup and Restore product to enable members of the Safeguard SECURITY_OSS_ADMINISTRATOR (SOA) group to back up and to restore files that are in restricted-access filesets. See the **initfilepriv** command.

File privileges are also removed from a file if the file is modified. Any changes to the file privileges on a file is audited. File privileges are inherited by child processes created using the **fork( )** function.

If the main executable of a process has a file privilege, then all user libraries and ordinary DLLs loaded into the process must also have that file privilege. Public DLLs and implicit DLLs do not need file privileges to be loaded into a process.

NFS client processes are not allowed to write to a file that has file privileges.

## RETURN VALUES

Upon successful completion, the **setfilepriv( )** function If the **setfilepriv( )** function call fails, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **setfilepriv( )** function sets **errno** to the corresponding value:

[EACCES] One of these conditions occured:

- Search permission was denied on a component of the pathname prefix
- The file does not exist
- The process attempted to access a Guardian subvolume with a reserved filename beginning with **ZYQ** or a file within such a subvolume

[EPERM] One of these conditions occured:

- The effective user ID was not a member of the Safeguard SECURITY-PRV-ADMINISTRATOR (SPA) group.
- The process attempted to set file privileges on an OSS file where the set-user-ID or set-group-ID bit of the file mode was already set.
- The process attempted to set file privileges on a Guardian file where the PROGID bit was already set.
- The process attempted to set file privileges on a file already opened for writing.

[EINVAL] The value specified for *fileprivs* is not valid.

[EIO] A physical input or output error occurred. The device where the file is stored might be in the down state, or both processors that provide access to the device might have failed.

[ENOSUP] The file specified by *path* either resides in a fileset that does not support file privileges, or is a file type that does not support file privileges (such as a directory or an AF_UNIX socket).

[EROFS] The file resides on a read-only fileset.

## RELATED INFORMATION

Commands: **getfilepriv(1), initfilepriv(1)**, **setfilepriv(1)**.

Functions: **chmod(2)**, **chown(2)**, **exec(2)**, **fork(2)**, **open(2)**, **stat(2)**.

## STANDARDS CONFORMANCE

This function is an HP extension.

## NAME

**setgid** - Sets the group ID of the calling process

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsecsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zsecdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/ysecdll**

## SYNOPSIS

```
#include <sys/types.h>    /* Optional except for POSIX.1 */
#include <unistd.h>

int setgid(
        gid_t gid);
```

## PARAMETERS

*gid* Specifies the new group ID.

## DESCRIPTION

The **setgid( )** function sets the real group ID, effective group ID, and saved-set-group ID of the calling process to the value specified by the *gid* parameter.

If the process does not have appropriate privileges but the *gid* parameter is equal to the real group ID or the saved-set-group ID, the **setgid( )** function sets the effective group ID to *gid*; the real group ID and saved-set-group ID remain unchanged.

If the calling process has appropriate privileges, the real group ID and saved-set-group ID are set to *gid* along with the effective group ID.

The group list of the calling process remains unchanged.

The value of *gid* must be in the range 0 through 65535.

## RETURN VALUES

Upon successful completion, the **setgid( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **setgid( )** function sets **errno** to the corresponding value:

[EINVAL] The value of the *gid* parameter is invalid or out of range.

[EPERM] The process lacks appropriate privileges and the *gid* parameter does not match the real group ID or the saved-set-group ID.

## RELATED INFORMATION

Functions: **exec(2)**, **getgid(2)**, **setuid(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- A process without appropriate privileges can set the effective group ID if the new effective group ID matches a group ID in the group list of the process.

## NAME

**setgroups -** Sets the group list of the calling process

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsecsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zsecdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/ysecdll**

## SYNOPSIS

```
#include <sys/types.h>  /* optional except for POSIX.1 */
#include <unistd.h>

int setgroups(
        int ngroups,
        const gid_t *gidset);
```

## PARAMETERS

*ngroups* Indicates the number of entries in the array pointed to by the *gidset* parameter. Must be no greater than the value of NGROUPS_MAX, which is defined in the <limits.h> header file.

*gidset* Points to the array of the group list that is to be set for the calling process.

## DESCRIPTION

The **setgroups( )** function sets the group list of the calling process according to the array pointed to by the *gidset* parameter. The *ngroups* parameter indicates the number of entries in the array, and must not exceed the value of NGROUPS_MAX, which is defined in the <limits.h> header file.

The calling process must have the appropriate privileges to use this function.

## NOTES

This function is supported on systems running J06.07 and later J-series RVUs, H06.18 and later H-series RVUs, and G06.33 and later G-series RVUs.

## RETURN VALUES

Upon successful completion, the **setgroups( )** function returns the value 0 (zero). Otherwise, the value **-**1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occur, the **setgroups( )** function sets **errno** to the corresponding value:

[EINVAL] The value of the *ngroups* parameter is greater than the value of NGROUPS_MAX or is not a positive number, or an entry in the array pointed to by the *gidset* parameter is not a valid group ID.

[EPERM] The process lacks appropriate privileges.

## RELATED INFORMATION

Functions: **getgroups(2)**, **initgroups(3)**.

## STANDARDS CONFORMANCE

This function conforms to the Application Environment Specification (AES) and the System V Interface Definition, version 3 (SVID3).

## NAME

**setpgid -** Sets the process group ID for job control

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>

int setpgid(
        pid_t pid,
        pid_t pgid);
```

## PARAMETERS

*pid* Specifies the process whose process group ID is to be changed.

*pgid* Specifies the new process group ID.

## DESCRIPTION

The **setpgid( )** function is used either to join an existing process group or to create a new process group within the session of the calling process. The process group ID of a session leader will not change.

The process group ID of the process designated by the *pid* parameter is set to the value of the *pgid* parameter. If *pid* is 0 (zero), the process ID of the calling process is used. If *pgid* is 0 (zero), the process ID of the indicated process is used.

### Use From the Guardian Environment

Calls to **setpgid( )** from Guardian processes are not successful. Such calls return an **errno** value of [ENOTOSS].

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. If the call was unsuccessful and initiated by an OSS process, the value -1 is returned and **errno** is set to indicate the error. If unsuccessful and initiated by a Guardian process, Guardian trap number 5 is set.

## ERRORS

If any of the following conditions occurs, the **setpgid( )** function sets **errno** to the corresponding value:

[EACCES] The value of the *pid* parameter matches the process ID of a child process of the calling process, and the child process has successfully executed one of the **exec**, **tdm_exec**, or **tdm_spawn** set of functions.

[EINVAL] One of the following conditions exists:

- The value of the *pgid* parameter is less than 0 (zero).
- The value of the *pgid* parameter is not a valid OSS process ID.
- Either the *pgid* or *pid* parameter is out of range.

[ENOTOSS] The calling process is not an OSS process. The requested operation is not supported from the Guardian environment.

[EPERM] One of the following conditions exists:

- The process indicated by the *pid* parameter is a session leader.
- The value of the *pid* parameter matches the OSS process ID of a child process of the calling process, and the child process is not in the same session as the calling process.
- The value of the *pgid* parameter is valid, but it does not match the OSS process ID of the process indicated by the *pid* parameter, and there is no process with a process group ID that matches the value of *pgid* in the same session as the calling process.

[ESRCH] The value of the *pid* parameter does not match the OSS process ID of the calling process or of a child process of the calling process.

## RELATED INFORMATION

Functions: **exec(2)**, **getpgrp(2)**, **setsid(2)**, **tcsetpgrp(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** value [ENOTOSS] can be returned.

## NAME

**setpgrp -** Sets the process group ID

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/types.h>**
**#include <unistd.h>**

**pid_t setpgrp(void);**

## DESCRIPTION

The **setpgrp( )** function creates a new session when the calling process is not a process group leader. The calling process then becomes the session leader of this session and the process leader of a new process group, and it has no controlling terminal. The process group ID of the calling process is set equal to its OSS process ID. The calling process becomes the only process in the new process group and the only process in the new session.

If the calling process is already a session group leader, the call fails and **errno** is set to [EPERM].

### Use From the Guardian Environment

Calls to **setpgrp( )** from Guardian processes are not successful. Such calls return an **errno** value of [ENOTOSS].

## NOTES

The **setpgrp( )** function is equivalent to the **setsid( )** function.

## RETURN VALUES

Upon successful completion, the value of the new process group ID is returned. If the call was unsuccessful and initiated by an OSS process, the value -1 is returned and **errno** is set to indicate the error. If unsuccessful and initiated by a Guardian process, Guardian trap number 5 is set.

## ERRORS

If any of these conditions occurs, the **setpgrp( )** function sets **errno** to the corresponding value:

[ENOTOSS] The calling process is not an OSS process. The requested operation is not supported from the Guardian environment.

[EPERM] One of these conditions exists:

- The calling process is already the process group leader.
- The process group ID of a process other than the calling process matches the OSS process ID of the calling process.

## RELATED INFORMATION

Functions: **setpgid(2)**, **setsid(2)**.

## STANDARDS CONFORMANCE

The following **errno** values are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [ENOTOSS] and [EPERM] can be returned.

## NAME

**setregid** - Sets the real and effective group IDs

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsecsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zsecdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/ysecdll**

## SYNOPSIS

```
#include <unistd.h>

int setregid(
        gid_t rgid,
        gid_t egid );
```

## PARAMETERS

*rgid* Specifies the new real group ID.

*egid* Specifies the new effective group ID.

## DESCRIPTION

The **setregid( )** function sets the real group ID and the effective group ID of the current process to the values specified by the *rgid* and *egid* parameters, respectively.

A process with appropriate privileges can set either group ID to any value.

A process without appropriate privileges can:

- Set the real group ID to the saved-set group ID used with an **execv( )** function call
- Set the effective group ID to the saved-set group ID used with an **execv( )** function call
- Set the effective group ID to the real group ID

Supplementary group IDs remain unchanged after a call to this function.

Supplying a value of -1 for either the real or effective group ID forces the system to substitute the current group ID in place of the -1 value.

## NOTES

The **setregid( )** function can be called only by native processes.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occur, the **setregid( )** function sets **errno** to the corresponding value:

[EINVAL] The value of the *rgid* or *egid* parameter is invalid or out of range.

[EPERM] The calling process does not have appropriate privileges and a change requiring appropriate privileges was specified.

## RELATED INFORMATION

Functions: **execv(2)**, **getgid(2)**, **setgid(2)**.

**STANDARDS CONFORMANCE**

The following are HP extensions to the XPG4 Version 2 specification:

- If both the real and effective group IDs are changed so that they differ from each other and from the saved-set group ID, then the saved-set group ID is set to the value of the effective group ID.
- A process without appropriate privileges can set the effective group ID if the new effective group ID matches a group ID in the group list of the process.

## NAME

**setreuid -** Sets the real and effective user IDs

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsecsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zsecdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/ysecdll**

## SYNOPSIS

```
#include <unistd.h>

int setreuid(
        uid_t ruid,
        uid_t euid );
```

## PARAMETERS

*ruid* Specifies the new real user ID.

*euid* Specifies the new effective user ID.

## DESCRIPTION

The **setreuid( )** function sets the real user ID and effective user ID of the current process to the values specified by the *ruid* and *euid* parameters, respectively. If *ruid* or *euid* has a value of -1, the current user ID (UID) is used by the system.

A process with appropriate privileges can set either ID to any value.

A process without appropriate privileges:

- Can set the effective user ID only if the *euid* parameter is equal to either the real, effective, or saved user ID of the process.
- Cannot set the real user ID.

Changing the effective user ID sets the process access ID (PAID) to the value of the effective user ID.

## NOTES

The **setreuid( )** function can be called only by native processes.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occur, the **setreuid( )** function sets **errno** to the corresponding value:

[EINVAL] The value of the *ruid* or *euid* parameter is invalid or out of range.

[EPERM] The current process does not have appropriate privileges and a change requiring appropriate privileges was specified.

## RELATED INFORMATION

Functions: **getuid(2)**, **setuid(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- If both the real and effective user IDs are changed so that they differ from each other and from the saved-set user ID, then the saved-set user ID is set to the value of the effective user ID.
- A process without appropriate privileges cannot set the real user ID.

**NAME**

**setsid -** Creates a new session and sets the process group ID

**LIBRARY**

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

**SYNOPSIS**

**#include <sys/types.h>** /* optional except for POSIX.1 */
**#include <unistd.h>**

**pid_t setsid(void);**

**DESCRIPTION**

The **setsid( )** function creates a new session when the calling process is not a process group leader. The calling process then becomes the session leader of this session, the process leader of a new process group, and has no controlling terminal. The process group ID of the calling process is set equal to its process ID. The calling process becomes the only process in the new process group and the only process in the new session.

**Use From the Guardian Environment**

Calls to **setsid( )** from Guardian processes are not successful. Such calls return an **errno** value of [ENOTOSS].

**RETURN VALUES**

Upon successful completion, the value of the new process group ID is returned. If the call was unsuccessful and initiated by an OSS process, the value -1 is returned and **errno** is set to indicate the error. If unsuccessful and initiated by a Guardian process, Guardian trap number 5 is set.

**ERRORS**

If any of the following conditions occurs, the **setsid( )** function sets **errno** to the corresponding value:

[ENOTOSS] The calling process is not an OSS process. The requested operation is not supported from the Guardian environment.

[EPERM] One of the following conditions exists:

- The calling process is already the process group leader.
- The process group ID of a process other than the calling process matches the OSS process ID of the calling process.

**RELATED INFORMATION**

Functions: **getpid(2)**, **setpgid(2)**.

**STANDARDS CONFORMANCE**

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** value [ENOTOSS] can be returned.

## NAME

**setsockopt -** Sets socket options

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <sys/socket.h>**
[**#include <netinet/in.h>** ]
[**#include <netinet/in6.h>** ]
[**#include <netdb.h>** ]
[**#include <netinet/tcp.h>**] Required for TCP protocol level

**int setsockopt(**
**int** *socket***,**
**int** *level***,**
**int** *option_name***,**
**const void ****option_value***,**
**socklen_t** *option_len***);**

## PARAMETERS

*socket* Specifies the file descriptor for the socket.

*level* Specifies the protocol level at which the option resides. The following values can be specified for the *level* parameter in an OSS application program:

**IPPROTO_IPV6**
Set IP protocol-level options defined for an Internet Protocol version 6 (IPv6) socket

**IPPROTO_IP** Set IP protocol-level options defined for an Internet Protocol version 4 (IPv4) socket

**IPPROTO_TCP**
Set TCP protocol-level options defined for a socket

**SOL_SOCKET**
Set socket-level protocol options defined for a socket

To set options at other levels, supply the appropriate protocol number for the protocol controlling the option. Valid protocol numbers can be found in **/etc/protocols**.

*option_name* Specifies the option to set. The *option_name* parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation.

The **sys/socket.h** header file defines the socket-level options. Additional header files are required for options at other levels.

The socket-level options can be enabled or disabled.

The **IPPROTO_IPV6** (IP protocol-level IPv6) options are:

**IPV6_JOIN_GROUP**
Enables the receipt of IPv6 multicast UDP datagrams for a specific group.

**IPV6_LEAVE_GROUP**
Disables the receipt of IPv6 multicast UDP datagrams for a specific group.

**IPV6_MULTICAST_IF**
Specifies the interface (subnet) to use for outbound multicast UDP datagrams. *option_value* is an **unsigned int**.

**IPV6_MULTICAST_HOPS**
Specifies the hop limit for outbound multicast UDP datagrams. *option_value* is an **int** that is either:

- Between 0 and 255 to indicate the maximum number of hops allowed
- -1 to indicate a limit of 255 hops

The default maximum number of hops allowed is 1. All other values cause an error and **errno** is set to [EINVAL].

**IPV6_MULTICAST_LOOP**
Enables or disables multicast messages sent to loopback for applications that have joined the same group on the same interface. This option is enabled by default. *option_value* is an **unsigned int**.

**IPV6_UNICAST_HOPS**
Specifies the hop limit for outbound unicast UDP datagrams. *option_value* is an **int** that is either:

- Between 0 and 255 to indicate the maximum number of hops allowed
- -1 to indicate that the default value should be used

All other values cause an error and **errno** is set to [EINVAL].

**IPV6_V6ONLY**
Specifies that **AF_INET6** sockets are restricted to IPv6-only communication.

The **IPPROTO_IP** (IP protocol-level IPv4) options are:

**IP_OPTIONS** Sets IP options for each outgoing packet. The *option_value* parameter is a pointer to a list of IP options and values that conforms with RFC 791.

**IP_ADD_MEMBERSHIP**
Enables the receipt of IP multicast UDP datagrams for a specific group.

**IP_DROP_MEMBERSHIP**
Disables the receipt of IP multicast UDP datagrams for a specific group.

**IP_MULTICAST_IF**

Specifies the interface (subnet) to use for outbound multicast UDP datagrams. *option_value* is an **unsigned int**.

**IP_MULTICAST_TTL**

Specifies the hop limit for outbound multicast UDP datagrams. *option_value* is an **int** that is either:

- Between 0 and 255 to indicate the maximum number of hops allowed
- -1 to indicate a limit of 255 hops

The default maximum number of hops allowed is 1. All other values cause an error and **errno** is set to [EINVAL].

**IP_MULTICAST_LOOP**

Enables or disables multicast messages sent to loopback for applications that have joined the same group on the same interface. This option is enabled by default. *option_value* is an **unsigned int**.

The **SOL_SOCKET** (socket-level protocol) options are:

**SO_BROADCAST**

Enables or disables sending of broadcast messages. The default value used when the socket is created is 0 (zero), which disables the option.

This option is valid only for **AF_INET** or **AF_INET6** datagram (UDP) sockets. If this option is specified for sockets of other types, the function call fails and **errno** is set to [ENOPROTOOPT].

*option_value* takes an **int** value. Specifying any nonzero value enables broadcast messages.

**SO_DEBUG** Enables or disables recording of debugging information in the underlying protocol modules. The default value used when the socket is created is 0 (zero), which disables the option.

This option is valid only for **AF_INET** or **AF_INET6** sockets. If this option is specified for sockets of other types, the function call fails and **errno** is set to [ENOPROTOOPT].

*option_value* takes an **int** value. Specifying a nonzero value enables recording of debugging information.

**SO_DONTROUTE**

Specifies whether outgoing messages should bypass the standard routing facilities and be directed to the appropriate network interface, according to the destination address. The default value used when the socket is created is 0 (zero), which indicates the use of standard routing.

This option is valid only for **AF_INET** or **AF_INET6** sockets. If this option is specified for sockets of other types, the function call fails and sets **errno** to [ENOPROTOOPT].

*option_value* takes an **int** value. Specifying any nonzero value bypasses normal routing.

**SO_KEEPALIVE**
Specifies whether to keep connections active by enabling the periodic transmission of messages on a connected socket. The default value used when the socket is created is 0 (zero), which indicates that no periodic messages are sent.

This option is valid only for **AF_INET** or **AF_INET6** sockets. If this option is specified for sockets of other types, the function call fails and sets **errno** to [ENOPROTOOPT].

*option_value* takes an **int** value. Specifying any nonzero value causes periodic transmission of messages.

**SO_LINGER** Controls whether the system attempts to deliver unsent data that is queued when a call to the **close( )** function occurs.

This option is valid only for **AF_INET** or **AF_INET6** sockets. If this option is specified for sockets of other types, the function call fails and **errno** is set to [ENOPROTOOPT].

*option_value* takes a **struct linger** value, as defined in the **sys/socket.h** header file. However, regardless of the option value, **SO_LINGER** is always enabled.

**SO_OOBINLINE**
Specifies whether received out-of-band data (data marked urgent) is queued with other data. The default value used for the option when the socket is created is 0 (zero), which indicates that urgent data is delivered separately.

This option is valid only for **AF_INET** or **AF_INET6** sockets. If this option is specified for sockets of other types, the function call fails and sets **errno** to [ENOPROTOOPT].

*option_value* takes an **int** value. Specifying any nonzero value causes out-of-band data to remain queued with other data.

**SO_RCVBUF** Sets the receive buffer size in bytes. The default value used for the option when the socket is created is 8K bytes.

This option is valid only for **AF_INET** or **AF_INET6** sockets. If this option is specified for sockets of other types, the function call fails and **errno** is set to [ENOPROTOOPT].

*option_value* takes an **int** value. Specifying a 0 (zero) value, a negative value, or a value greater than 262144 causes the function call to fail with **errno** set to [EINVAL].

**SO_REUSEADDR**
Specifies whether the rules used in validating addresses supplied by a **bind( )** function call should allow reuse of local addresses. The default value used for the option when the socket is created is 0 (zero), which indicates that addresses should not be reused.

*option_value* takes an **int** value. Specifying a nonzero value permits addresses to be reused.

**SO_REUSEPORT**
Specifies whether the rules used in validating ports supplied by a **bind( )** function call should allow reuse of local ports. The default value used for the option when the socket is created is 0 (zero), which indicates that ports should not be reused.

This option is valid only for UDP ports.

This option takes an **int** value. Specifying a nonzero value permits ports to be reused.

**SO_SNDBUF** Sets the send buffer size in bytes. The default value used for the option when the socket is created is 8K bytes.

This option is valid only for **AF_INET** or **AF_INET6** sockets. If this option is specified for sockets of other types, the function call fails and sets **errno** to [ENOPROTOOPT].

*option_value* takes an **int** value. Specifying a 0 (zero) value, a negative value, or a value greater than 262144 causes the function call to fail with **errno** set to [EINVAL].

The **IPPROTO_TCP** (TCP protocol-level) options are:

**TCP_MAXRXMT**
Sets the maximum retransmission timeout value in multiples of 500 milliseconds.

*option_value* takes an **int** value. Valid values are in the range 1 through 60. The value specified for this option should be greater than or equal to the value used for the **TCP_MINRXMT** option.

**TCP_MINRXMT**
Sets the minimum retransmission timeout value in multiples of 500 milliseconds.

*option_value* takes an **int** value. Valid values are in the range 1 through 2400. The value specified for this option should be less than or equal to the value used for the **TCP_MAXRXMT** option.

**TCP_NODELAY**
Specifies whether data packets are buffered before transmission.

*option_value* takes an **int** value. A nonzero value indicates that data packets should not be buffered. A 0 (zero) value indicates that buffering should occur.

**TCP_RXMTCNT**
Sets the maximum retransmission count.

*option_value* takes an **int** value. Valid values are in the range 1 through 12. When the value specified for this option is multiplied by the value used for the **TCP_MAXRMT** option and the result is less than the value used for **TCP_TOTRXMTVAL**, the TCP connection will be dropped before the **TCP_TOTRXMTVAL** value is reached.

**TCP_SACKENA**
Specifies whether TCP selective acknowledgments are enabled.

*option_value* takes an **int** value. A nonzero value indicates that selective acknowledgments are enabled. A 0 (zero) value indicates that selective acknowledgments should not be used.

**TCP_TOTRXMTVAL**
Sets the total maximum retransmission duration in multiples of 500 milliseconds. Once the duration is reached, the TCP cpnnection is dropped.

*option_value* takes an **int** value. Valid values are in the range 1 through 28800. When the value specified for the **TCP_RXMTCNT** option is multiplied by the value used for the **TCP_MAXRMT** option and the result is less than the value used for **TCP_TOTRXMTVAL**, the TCP connection will be dropped before the **TCP_TOTRXMTVAL** value is reached.

Options at other protocol levels vary in format and name.

*option_value* Points to the buffer containing the appropriate option value. For options that can be classified as disabled or enabled, a value of 0 (zero) indicates that the option should be disabled and a value of 1 indicates that the option should be enabled.

*option_len* Contains the size of the buffer pointed to by the *option_value* parameter.

## DESCRIPTION

The **setsockopt( )** function sets options associated with a socket. Options can exist at multiple protocol levels. The **SO_*** options are always present at the uppermost socket level.

The **setsockopt( )** function provides an application program with the means to control socket communication. An application program can use the **setsockopt( )** function to enable debugging at the protocol level, allocate buffer space, control time-outs, or permit socket data broadcasts. The **sys/socket.h** header file defines all the **SO_*** options available to the **setsockopt( )** function.

If your application uses the Cluster I/O Protocols (CIP) subsystem, options for this function might not be supported or might result in behaviors that are different from those described in this reference page. For more information about the Cluster I/O Protocols, see the *Cluster I/O Protocols (CIP) Configuration and Management Manual*.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **setsockopt( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **setsockopt( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EINVAL] One of the following conditions exists:

- The specified option is not valid at the specified socket level.
- The socket has been shut down.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time might succeed.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOPROTOOPT]
The specified option is not supported by the protocol used by the socket.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

## RELATED INFORMATION

Functions: **bind(2)**, **endprotoent(3)**, **getprotobynumber(3)**, **getprotoent(3)**, **getsockopt(2)**, **setprotoent(3)**, **socket(2)**, **socketpair(2)**.

## STANDARDS CONFORMANCE

The HP implementation does not return the **errno** value [ENOSR].

The following are HP extensions to the XPG4 specification:

- Nonzero values other than 1 can be used to set Boolean options.
- The **SO_DONTROUTE** and **SO_REUSEPORT** options are supported.
- The **errno** value [ECONNRESET] can be returned when the transport-provider process is unavailable.
- Some of the documented uses of the **errno** value [ENOPROTOOPT] are not described in the specification.

## NAME

**setuid** - Sets the user ID of the calling process

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsecsrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zsecdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/ysecdll**

## SYNOPSIS

```
#include <sys/types.h>    /* optional except for POSIX.1 */
#include <unistd.h>

int setuid(
        uid_t uid);
```

## PARAMETERS

*uid* Specifies the new user ID.

## DESCRIPTION

When invoked by processes with appropriate privileges, the **setuid( )** function sets the real user ID, effective user ID, and saved-set-user ID of the calling process to the value of the *uid* parameter.

To change the real user ID, the effective user ID, and the saved-set-user ID, the calling process must have appropriate privileges. If the process does not have appropriate privileges but the *uid* parameter is equal to the real user ID or the saved-set-user ID, the **setuid( )** function sets the effective user ID to *uid*; the real user ID and saved-set-user ID remain unchanged.

The value of *uid* must be in the range 0 through 65535.

## NOTES

Changing the effective user ID sets the operating system process access ID (PAID) to the value of the effective user ID.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **setuid( )** function sets **errno** to the corresponding value:

[EINVAL] The *uid* parameter is out of range.

[EPERM] The process lacks appropriate privileges, and the *uid* parameter does not match the real user ID or the saved-set-user ID.

## RELATED INFORMATION

Functions: **exec(2)**, **getuid(2)**.

## NAME

**shmat -** Attaches a shared memory segment to the address space of the calling process

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
H-series and J-series native Guardian processes: $SYSTEM.ZDLL*nnn*.ZOSSKDLL
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

## SYNOPSIS

**#include <sys/shm.h>**

**void *shmat(**
**int** *shmid***,**
**const void** **shmaddr***,**
**int** *shmflag***);**

## PARAMETERS

*shmid*
Specifies the identifier for the shared memory segment. The identifier is usually returned by a previous call to the **shmget( )** function.

*shmaddr*
Specifies the virtual address at which the process wants to attach the shared memory segment. The process can also specify 0 (zero) to have the system select an appropriate address.

*shmflag*
Specifies the following attach flags:

**SHM_RND** If the *shmaddr* parameter is not a null pointer, the system rounds off the address, if necessary.

**SHM_RDONLY**
The segment is attached for read-only access.

## DESCRIPTION

The **shmat( )** function attaches the shared memory segment identified by the *shmid* parameter to the virtual address space of the calling process. For the *shmaddr* parameter, the process can specify an explicit address, or it can pass a NULL pointer (zero) to have the system select the address. If *shmaddr* is nonzero and (*shmflag* & **SHM_RND**) is not zero, the system rounds down the specified address. For detailed information, see "Shared Memory Segment Alignment."

The segment is attached for reading if (*shmflag* & **SHM_RDONLY**) is not zero and the calling process has read permission. If (*shmflag* & **SHM_RDONLY**) is 0 (zero) and the calling process has read and write permission, the segment is attached for reading and writing.

Memory can be shared only within the same processor.

Shared memory uses operating system flat segments (permanently mapped shareable data segments). Refer to the *Guardian Programmer's Guide* for more information about flat segments.

### Address Range

An application that is using the shared memory functions **shmat( )** and **shmdt( )** to manage a range of virtual addresses should use only these functions to manipulate the range.

The valid range of addresses for the *shmaddr* parameter can change from one release to the next. Programs should not contain hard-coded addresses.

### Shared Segment Memory Alignment

On servers running J06.12 or later J-series RVUs, or H06.23 or later H-series RVUs:

- If **shmaddr** is nonzero and rounding is specified, the specified address is rounded to a multiple of 4 MB.
- If **shmaddr** is nonzero and rounding is not specified, the specified address must be a multiple of 16 KB.
- If **shmaddr** is zero, the system chooses an address that is a multiple of at least 16 KB. The SHMLBA constant is irrelevant.

On servers running earlier J-series RVUs, earlier H-series RVUs, or G-series RVUs:

- If **shmaddr** is nonzero and rounding is specified, the specified address is rounded to a multiple of 32 MB.
- If **shmaddr** is nonzero and rounding is not specified, the specified address must be a multiple of 32 KB.
- If **shmaddr** is zero, the system chooses an address that is a multiple of at least 16 KB (but not necessarily a multiple of 32 MB). The SHMLBA constant is 32 MB.

### Number of Shared Segments

On servers running J06.12 or later J-series RVUs, or H06.23 or later H-series RVUs, there is no configured limit to the number of OSS shared memory segments that can be attached by one process. The number of OSS shared memory segments that can be attached is limited by system resources only.

On servers running earlier J-series RVUs, earlier H-series RVUs, or G-series RVUs, a process can attach no more than 13 segments at one time.

### Propagation During Process Creation

Segments attached to a parent process are also attached to a child process created by the **fork( )** or **tdm_fork( )** function.

Segments attached to a parent process are not propagated by a call to:

- Any of the **exec** or **tdm_exec** sets of functions
- Any of the **tdm_spawn** or PROCESS_SPAWN_ set of functions

The resulting child process has no attached shared memory segments.

### Use From the Guardian Environment

On servers running J06.12 or later J-series RVUs, or H06.23 or later H-series RVUs, Guardian processes can attach shared memory segments. Permissions are handled in the same way for both OSS and Guardian processes.

If called from a Guardian process on servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs, this function call fails and **errno** is set to the value of [ENOTOSS].

## NOTES

The shared memory identifier, *shmid*, is not the Guardian environment **segid** value or segment identifier.

Programs should not be written to depend upon the maximum number of attached shared segments. This limit is subject to change.

Refer to the SEGMENT_ALLOCATE_ procedure description in the *Guardian Procedure Calls Reference Manual* for more information about segment limits.

## RETURN VALUES

Upon successful completion, the **shmat( )** function increments the value of the **shm_nattch** field in the structure associated with the shared memory identifier of the attached shared memory segment. The starting address for the attached segment is returned.

Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **shmat( )** function sets **errno** to the corresponding value and does not attach the shared memory segment:

[EACCES] The calling process does not have access permission for the requested operation.

[EINVAL] One of the following is true:

- The *shmid* parameter does not specify a valid shared memory identifier.
- All of these conditions are true:
  - The *shmaddr* parameter is not a null pointer
  - Rounding is not specified: (*shmflag* & **SHM_RND**) is 0 (zero).
  - There is inadequate virtual address available in the process to allocate the requested segment at the specified or rounded or default address.

[EMFILE] On servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs, an attempt to attach a shared memory segment exceeded the maximum number of attached segments allowed for any one process.

[ENOMEM] There was not enough data space available to attach the shared memory segment and allocate the associated low-level data structures.

[ENOTOSS] The calling process is not an OSS process. The requested operation cannot be performed from the Guardian environment on servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs.

## RELATED INFORMATION

Commands: **ipcrm(1)**, **ipcs(1)**.

Functions: **exec(2)**, **_exit(2)**, **fork(2)**, **shmctl(2)**, **shmdt(2)**, **shmget(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**.

## STANDARDS CONFORMANCE

The **SHMLBA** constant is used for a nontraditional value.

The following are HP extensions to the XPG4 Version 2 specification:

- On servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs, the **errno** value [ENOTOSS] can be returned if this function is called from a Guardian process.

## NAME

**shmctl -** Performs shared memory control operations

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
H-series and J-series native Guardian processes: $SYSTEM.ZDLL*nnn*.ZOSSKDLL
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

## SYNOPSIS

**#include <sys/shm.h>**

**int shmctl(**
    **int** *shmid***,**
    **int** *cmd***,**
    **struct shmid_ds *****buf***);**

## PARAMETERS

*shmid*
Specifies the shared memory identifier for the segment.

*cmd*
Specifies the type of operation. The possible values for *cmd* and the operations they perform are as follows:

**IPC_RMID**
Removes the shared memory identifier and deallocates its associated **shmid_ds** structure.

This is a restricted operation. The effective user ID of the calling process either must have appropriate privileges or must be equal to the value of the creator's user ID (**shm_perm.cuid** field) or the owner's user ID (**shm_perm.uid** field) in the associated **shmid_ds** structure.

**IPC_SET**
Sets the shared memory identifier by copying selected values in the structure specified by the *buf* parameter into the corresponding fields in the **shmid_ds** structure associated with the shared memory identifier.

This is a restricted operation. The effective user ID of the calling process either must have appropriate privileges or must be equal to the value of the creator's user ID (**shm_perm.cuid** field) or the owner's user ID (**shm_perm.uid** field) in the associated **shmid_ds** structure.

**IPC_STAT**
Queries the shared memory identifier by copying the contents of its associated **shmid_ds** structure into the structure specified by the *buf* parameter. The calling process must have read access to the segment.

*buf*
Specifies the address of a **shmid_ds** structure. This structure is used only with the **IPC_STAT** and **IPC_SET** values of the *cmd* parameter. With **IPC_STAT**, the results of the query are copied to this structure. With **IPC_SET**, the values in this structure are used to set certain fields in the **shmid_ds** structure associated with the shared memory identifier. In either case, the calling process must have allocated the structure before making the call.

## DESCRIPTION

The **shmctl( )** function allows a process to query or set the contents of the **shmid_ds** structure associated with the specified shared memory identifier. It also allows a process to remove the shared memory identifier and its associated **shmid_ds** structure. The value of the *cmd* parameter determines which operation is performed.

The **IPC_SET** value of the *cmd* parameter uses the user-supplied contents of the *buf* structure to set the corresponding fields in the **shmid_ds** structure associated with the shared memory identifier. The fields are set as follows:

- The owner's user ID field (**shm_perm.uid**) is set as specified in the input.
- The owner's group ID field (**shm_perm.gid**) is set as specified in the input.
- The access modes field (**shm_perm.mode**) is set as specified in the low-order nine bits of the corresponding field in the input.

The **IPC_SET** and **IPC_RMID** values of the *cmd* parameter also update the **shm_perm.ctime** field to the current time.

### Use From the Guardian Environment

On servers running J06.12 or later J-series RVUs or H06.23 or later H-series RVUs, Guardian process also can get and share shared memory segments. The Guardian process PIN is reported in place of the process ID in the **shm_lpid** and **shm_cpid** members of the **shmid_ds** structure reported by the **shmctl( )** function. These data are for information only and cannot be passed to a function that requires an actual **pid_t** value.

If called from a Guardian process on servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs, the function call fails and **errno** is set to the value of [ENOTOSS].

## NOTES

The shared memory identifier, *shmid*, is not the Guardian environment **segid** value or segment identifier.

Programs should not be written to depend upon the maximum number of attached shared segments. This limit is subject to change.

Refer to the SEGMENT_ALLOCATE_ procedure in the *Guardian Procedure Calls Reference Manual* for more information about segment limits.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **shmctl( )** function sets **errno** to the corresponding value:

[EACCES] The *cmd* parameter is **IPC_STAT**, but the calling process does not have read permission.

[EFAULT] One of the following conditions exists:

- The *cmd* parameter is **IPC_STAT**, and either the *buf* structure is not in the address space of the process or the function cannot write into the *buf* structure.

- The *cmd* parameter is **IPC_SET**, and the *buf* structure is not in the address space of the process.

[EINVAL] One of the following conditions exists:

- The *shmid* parameter does not specify a valid shared memory identifier.
- The *cmd* parameter is not a valid command.

[ENOTOSS] The calling process is not an OSS process. The requested operation cannot be performed from the Guardian environment on servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs.

[EPERM] The *cmd* parameter is equal to either **IPC_RMID** or **IPC_SET**, and the calling process does not have the correct privileges.

## RELATED INFORMATION

Commands: **ipcrm(1)**, **ipcs(1)**.

Functions: **shmat(2)**, **shmdt(2)**, **shmget(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT] can be returned.
- On servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs, the **errno** value [ENOTOSS] can be returned if this function is called from a Guardian process.
- If the relevant action was performed by a Guardian process, the reported values of the **shm_lpid** and **shm_cpid** members of the **shmid_ds** structure are Guardian PIN values.

## NAME

**shmdt -** Detaches a shared memory segment

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zossksrl**
H-series and J-series native Guardian processes: $SYSTEM.ZDLL*nnn*.ZOSSKDLL
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/yosskdll**

## SYNOPSIS

**#include <sys/shm.h>**

**int shmdt(**
    **const void ****shmaddr***);**

## PARAMETERS

*shmaddr* Specifies the starting virtual address for the shared memory segment that is to be detached. This is the address returned by a previous **shmat( )** function call.

## DESCRIPTION

The **shmdt( )** function detaches the shared memory segment at the indicated address from the address space of the calling process.

### Address Range

An application that is using the shared memory functions **shmat( )** and **shmdt( )** to manage a range of virtual addresses should use only these functions to manipulate the range.

The valid range of addresses for the *shmaddr* parameter can change from one release to the next. Programs should not contain hard-coded addresses.

### Cleaning Up Shared Memory Identifiers

A shared memory identifier remains allocated until it is removed. An allocated shared memory identifier is not removed when the last process using it terminates. The user must remove allocated shared memory identifiers that are not attached to processes to avoid wasting shared memory resources.

The status of shared memory identifiers can be checked with the **ipcs** command. Shared memory identifiers can be removed using the **ipcrm** command. The associated shared memory segment and data structure are removed only after the final detach operation.

### Use From the Guardian Environment

On servers running J06.12 or later J-series RVUs or H06.23 or later H-series RVUs, Guardian processes can use the **shmdt( )** function to detach from shared memory segments.

If called from a Guardian process on servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs, the function call fails and **errno** is set to the value of [ENOTOSS].

## NOTES

The shared memory identifier is not the Guardian environment **segid** value or segment identifier.

Programs should not be written to depend upon the maximum number of attached shared segments. This limit is subject to change.

Refer to the SEGMENT_ALLOCATE_ procedure in the *Guardian Procedure Calls Reference Manual* for more information about segment limits.

## RETURN VALUES

Upon successful completion, the **shmdt( )** function returns the value 0 (zero). The shared memory segment is detached. The value of the **shm_nattch** field in the structure associated with the shared memory identifier in the shared memory table is decremented.

Otherwise, the **shmdt( )** function returns the value -1 and sets **errno** to indicate the error.

## ERRORS

If any of the following conditions occur, the **shmdt( )** function sets **errno** to the corresponding value and does not detach the shared memory segment:

[EINVAL] The *shmaddr* parameter does not specify the starting address of a shared memory segment.

[ENOTOSS] The calling process is not an OSS process. The requested operation cannot be performed from the Guardian environment on servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs.

## RELATED INFORMATION

Commands: **ipcrm(1)**, **ipcs(1)**.

Functions: **shmat(2)**, **shmctl(2)**, **shmget(2)**.

## STANDARDS CONFORMANCE

The following is a HP extension to the XPG4 Version 2 specification:

- On servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs, the **errno** value [ENOTOSS] can be returned if this function is called from a Guardian process.

## NAME

**shmget -** Creates a new shared memory segment or returns the identifier of an existing shared memory segment

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zosksrl**
H-series and J-series native Guardian processes: $SYSTEM.ZDLL*nnn*.ZOSSKDLL
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/yosskdll**

## SYNOPSIS

**#include <sys/shm.h>**

**int shmget(**
  **key_t** *key***,**
  **size_t** *size***,**
  **int** *shmflag***);**

## PARAMETERS

*key* — Specifies the key that identifies the shared memory segment. The **IPC_PRIVATE** key can be used to ensure the return of a new (unused) shared memory identifier.

*size* — Specifies the minimum number of bytes to allocate for the shared memory segment.

*shmflag* — Specifies the access mode value to use for the segment, logically ORed with the creation flag value to use for the segment.

The access mode value occupies the least-significant nine bits of the parameter. These bits can be set by logically ORing any of the following symbolic values defined in the **sys/stat.h** header file:

**S_IRGRP**
**S_IROTH**
**S_IRUSR**
**S_IRWXG**
**S_IRWXO**
**S_IRWXU**
**S_IWGRP**
**S_IWOTH**
**S_IWUSR**
**S_IXGRP**
**S_IXOTH**
**S_IXUSR**

Refer to the **chmod(2)** reference page for more information about the correct use of these symbolic values.

The following creation flag values are valid:

**IPC_CREAT** — If the key does not exist, the **shmget( )** function creates a shared memory identifier using the given key.

**IPC_CREAT | IPC_EXCL** — If the key already exists, the **shmget( )** function fails and returns an error notification.

## DESCRIPTION

The **shmget( )** function returns the shared memory identifier for the shared memory segment identified by the *key* parameter. If the *key* parameter already has a shared memory identifier associated with it and (*shmflag* **& IPC_CREAT**) is 0 (zero), that identifier is returned.

A new shared memory identifier, the associated shared memory table entry, and a new shared memory segment of at least *size* bytes are created when either of the following is true:

- The value **IPC_PRIVATE** is used for the *key* parameter.
- The *key* parameter does not already have a shared memory identifier associated with it, and (*shmflag* **& IPC_CREAT**) is not 0 (zero).

After creating a new shared memory identifier, the **shmget( )** function initializes the shared memory table entry associated with the identifier as follows:

- The creator's user ID field (**shm_perm.cuid**) and owner's user ID field (**shm_perm.uid**) are set equal to the effective user ID of the calling process.
- The creator's group ID field (**shm_perm.cgid**) and owner's group ID field (**shm_perm.gid**) are set equal to the effective group ID of the calling process.
- The least-significant nine bits of the access mode field (**shm_perm.mode**) are set equal to the least-significant nine bits of the *shmflag* parameter.
- The shared memory segment size field (**shm_segsz**) is set to the value of the *size* parameter.
- The following fields are all set to 0 (zero):
  - — **shm_lpid**, the process ID or PIN of the latest process that performed a **shmat( )**, **shmdt( )**, or **shmctl( )** operation
  - — **shm_nattch**, the number of processes that currently have this region attached
  - — **shm_atime**, the time of the last **shmat( )** operation
  - — **shm_dtime**, the time of the last **shmdt( )** operation
- The **shm_ctime** field is set equal to the current time. This field is updated when any of the following events occur:
  - — The shared memory identifier is created.
  - — The permissions for the shared memory segment are changed.
  - — The shared memory identifier is removed.
- The process ID of the process that created the shared memory identifier (the **shm_cpid** field) is set to the process ID or PIN of the calling OSS or Guardian process.

The shared memory identifier is used for the following purposes:

- It identifies a specific entry in the system-maintained shared memory table.
- It allows detection of references to a previously removed shared memory identifier.

- It allows detection of attempts to reference shared memory segments in other processors.

**Key Creation**

The key represents a user-designated name for a given shared memory segment. Keys are usually selected by calling the **ftok( )** function before calling the **shmget( )** function. The **ftok( )** function returns a key based on a path and an interprocess communications identifier. This key is then passed to the **shmget( )** function, which returns a shared memory identifier. The shared memory identifier is then used in calls to the **shmat( )** and **shmctl( )** functions.

**Uniqueness of Identifiers**

The system recycles no-longer-used shared memory identifiers after a long time elapses.

**Swap File**

A shared memory segment is backed by Kernel-managed swap space so that its data remains intact even when no processes include it in their virtual address space.

**Processor or Disk Process Failures**

If a processor fails, the following is lost:

- All information in the system-maintained shared memory table for that processor
- All shared memory segments for that processor
- All corresponding swap files

If the disk process controlling the swap space fails, the system monitor causes any process with the corresponding shared memory attached to terminate abnormally. Thereafter, a process cannot successfully call either the **shmget( )** or **shmat( )** function using the associated shared memory identifier. When either function is called, the shared memory segment and its identifier are removed from the system-maintained shared memory table.

**Valid Segment Sizes**

On servers running J06.12 or later J-series RVUs, or H06.23 or later H-series RVUs, there is no configured limit to the size of an OSS shared memory segement. The size of the OSS shared memory segment is limited by system resources only.

On servers running earlier J-series RVUs, earlier H-series RVUs, or G-series RVUs, a shared memory segment can contain up to 128 megabytes (MB).

**Number of Shared Segments and Identifiers**

The maximum number of shared memory identifiers is determined by the maximum number of processes allowed for the processor. This value cannot exceed the limit **SHMT_MAXENTRIES**, which is currently set to 1000.

**Cleaning Up Shared Memory Identifiers**

A shared memory identifier remains allocated until it is removed. An allocated shared memory identifier is not removed when the last process using it terminates. The user must remove allocated shared memory identifiers that are not attached to processes to avoid wasting shared memory resources.

The status of shared memory identifiers can be checked with the **ipcs** command. Shared memory identifiers can be removed using the **ipcrm** command. The associated shared memory segment and data structure are removed only after the final detach operation.

**Use by OSS and Guardian Processes**

The shared memory segments managed by the **shm*( )** functions are distinct from segments created by SEGMENT_ALLOCATE_ and related Guardian procedure calls. Both kinds of segments can be created and shared, but one kind cannot be shared with the other. The shared memory identifier, **shmid**, is not a Guardian **segid** value.

Both Guardian and OSS processes can call SEGMENT_ALLOCATE_ and related Guardian procedure calls: they can share segments as described for SEGMENT_ALLOCATE_.

On servers running J06.12 or later J-series RVUs or H06.23 or later H-series RVUs, both Guardian processes and OSS processes can call **shmget( )** and related functions; they can share segments as described for **shmget( )**.

On servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs, only OSS processes can call **shmget( )** and related functions; Guardian process calls to these functions fail and **errno** is set to the value of [ENOTOSS].

**NOTES**

The shared memory identifier is not the Guardian environment **segid** value or segment identifier.

Programs should not be written to depend upon the maximum number of attached shared segments. This limit is subject to change.

Refer to the SEGMENT_ALLOCATE_ procedure in the *Guardian Procedure Calls Reference Manual* for more information about segment limits.

**RETURN VALUES**

Upon successful completion, a nonnegative shared memory identifier is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **shmget( )** function sets **errno** to the corresponding value:

[EACCES] A shared memory identifier already exists for the *key* parameter, but operation permission as specified by the low-order nine bits of the *shmflag* parameter was not granted.

[EEXIST] A shared memory identifier already exists for the *key* parameter, but **IPC_CREAT** and **IPC_EXCL** were both set in the *shmflag* parameter.

[EINVAL] One of the following conditions is true:

- The value of the *size* parameter is less than the system-defined minimum or greater than the system-defined maximum.
- A shared memory identifier already exists for the *key* parameter, but the number of bytes allocated for the region is less than *size* and *size* is not equal to 0 (zero).

[ENOENT] A shared memory identifier does not exist for the *key* parameter, and **IPC_CREAT** was not set in the *shmflag* parameter.

[ENOMEM] An attempt was made to create a shared memory identifier and its associated shared memory table entry, but there was not enough physical or virtual memory available.

[ENOSPC] An attempt to create a new shared memory identifier exceeded the system limit on the maximum number of identifiers allowed.

[ENOTOSS] The calling process is not an OSS process. The requested operation cannot be performed from the Guardian environment on servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs.

## RELATED INFORMATION

Commands: **ipcrm(1)**, **ipcs(1)**.

Functions: **ftok(3)**, **shmat(2)**, **shmctl(2)**, **shmdt(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- On servers running J06.11 or earlier J-series RVUs, H06.22 or earlier H-series RVUs, or G-series RVUs, the **errno** value [ENOTOSS] can be returned if this function is called from a Guardian process.

## NAME

**shutdown -** Shuts down socket send and receive operations

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/socket.h>**

**int shutdown(**
    **int** *socket***,**
    **int** *how*
    **);**

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*how* Specifies the type of shutdown. The values are as follows:

**SHUT_RD** Disables further receive operations.

**SHUT_RDWR**
Disables further send and receive operations.

**SHUT_WR** Disables further send operations.

## DESCRIPTION

The **shutdown( )** function disables receive operations, send operations, or both on the specified socket.

## RETURN VALUES

Upon successful completion, the **shutdown( )** function returns the value 0 (zero). Otherwise, the value **-**1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **shutdown( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EINVAL] The value specified for the *how* parameter is not valid.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time may succeed.

[ENOMEM] Required memory resources were not available. A retry at a later time may succeed.

[ENOTCONN] The socket is not connected.

[ENOTSOCK] The *socket* parameter does not specify a socket.

## RELATED INFORMATION

Functions: **getsockopt(2)**, **read(2)**, **recv(2)**, **recvfrom(2)**, **recvmsg(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **setsockopt(2)**, **socket(2)**, **write(2)**.

## STANDARDS CONFORMANCE

The HP implementation does not return the **errno** value [ENOSR].

The following are HP extensions to the XPG4 specification:

- The **errno** value [ECONNRESET] can be returned.

## NAME

**sigaction -** Specifies the action to take upon delivery of a signal

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
[#include <spthread.h>]
#include <signal.h>

int sigaction(
        int signal,
        const struct sigaction *action,
        struct sigaction *o_action);
```

## PARAMETERS

*signal*
: Specifies the signal. The signal names are defined in the **signal.h** header file. The range of valid signals depends on the requested action.

*action*
: Points to a **sigaction** structure that describes the action to be taken upon receipt of the signal identified by the *signal* parameter.

*o_action*
: Points to a **sigaction** structure that returns the signal action data in effect before the call was made. For the signal action in effect at the time of the **sigaction( )** call to be returned, the *o_action* parameter must not be a null pointer.

## DESCRIPTION

The **sigaction( )** function allows the calling process to change or examine the action to be taken when a specific signal is delivered to the calling process.

Associated with every signal is a signal-dependent default action. The **sigaction( )** function can change this action by causing the receiving process to

- Ignore the delivery of a specific signal
- Restore the default action for a specific signal
- Invoke a signal-catching function (that is, "catch" the signal) in response to the delivery of a specific signal

See the **signal(4)** reference page for the defined signal names and details about the cause and default action of each defined signal.

Unless you are writing a Standard POSIX Threads application, omit the **spthread.h** header file.

### Use From the Guardian Environment

The **sigaction( )** function can be called from any G-series, H-series or J-Series native Guardian process. If called from a TNS or accelerated Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

### Specifying the Signal

The *signal* parameter specifies the signal. All values defined for signals in the **signal.h** header file are valid if the corresponding action is to restore the default action. All signals can be caught or ignored except the **SIGKILL**, **SIGSTOP**, and **SIGABEND** signals; these signals can neither be caught nor ignored.

### Specifying the Action

If the *action* parameter is not a null pointer, it points to a **sigaction** structure that describes the action to be taken on receipt of the signal specified in the *signal* parameter.

If the *o_action* parameter is not a null pointer, it points to a **sigaction** structure in which the signal action data in effect at the time of the **sigaction( )** call is returned.

If the *action* parameter is a null pointer, signal handling is unchanged; thus, the call can be used to inquire about the current handling of a given signal.

If the previous action for *signal* was established by the **signal( )** function (see the **signal(3)** reference page), the values of the fields returned in the structure pointed to by *o_action* are unspecified and should not be depended upon. In particular, *o_action*->**sa_handler** is not necessarily the same value passed to the **signal( )** function. However, if a pointer to the same structure is passed to a subsequent call to the **sigaction( )** function using the *action* parameter, the signal is handled in the same way as if the original call to **signal( )** were repeated.

The **sigaction** structure is as follows:

```
struct sigaction {
        sigset_t    sa_mask;
        void (*sa_handler)(int);
        int         sa_flags;
};
```

The action is ignored when *action* is set to the **SIG_DFL** value for a signal that cannot be caught or ignored.

### Specifying the Handler

The **sa_handler** field in the **sigaction** structure can have one of the following values, or it can point to a function:

**SIG_ABORT** Requests that the process terminate abnormally when the signal is delivered. This value is defined in the **signal.h** header file.

**SIG_DEBUG** Requests that the debugger be entered when the signal is delivered. This value is defined in the **signal.h** header file.

**SIG_DFL** Requests default action to be taken when the signal is delivered; this value is defined in the **signal.h** header file.

**SIG_IGN** Rquests that the signal have no effect on the receiving process; this value is defined in the **signal.h** header file.

A pointer to a function requests that the signal be caught; that is, the signal causes the signal-catching function to be called.

These actions are described in detail in the **signal(4)** reference page.

### Blocking Signals

The **sa_mask** field in the **sigaction** structure specifies additional signals to be blocked from delivery while the signal-catching function is executing. The system creates a new signal mask from the existing process signal mask, the **sa_mask** field, and the delivered signal itself. All the signals in the new signal mask are blocked from delivery while the signal-catching function is executing or until a call is made to the **sigprocmask( )**, **pthread_sigmask( )** (for standard POSIX threads), or **sigsuspend( )** function. If and when the signal-catching function returns normally, the original signal mask is restored, regardless of any modifications made by the **sigprocmask( )** or **pthread_sigmask( )** function since the signal-catching function was invoked.

The **SIGKILL**, **SIGSTOP**, and **SIGABEND** signals cannot be blocked. If a program attempts to block any of these signals, the system removes them from the signal mask without generating an error. For example, if a call to **sigaction( )** tries to block the **SIGKILL** signal and then a subsequent call returns the signal-handling information in the structure pointed to by the *o_action* parameter, the returned mask is not the same mask that the original call passed in; the difference is that the returned mask does not include the **SIGKILL** signal.

### Specifying Options

If the **sigaltstack( )** function is used to specify an alternate signal stack for a user signal handler, and the alternate signal stack is registered and enabled, then all user signal handlers run on the alternate signal stack.

Unless you are using the Standard POSIX Threads library, although the **SA_ONSTACK** flag has no effect in the Guardian environment, to allow code portability, the **SA_ONSTACK** flag will be recognized on systems running J06.10 or later RVUs or H06.21 or later RVUs, if the **SA_ONSTACK_COMPATIBILITY** feature test macro is set. You should NOT use the **SA_ONSTACK** flag and the **SA_ONSTACK_COMPATIBILITY** feature test macro in a threaded application that uses the Standard POSIX Threads library. Use of these two options with the Standard POSIX Threads library can result in undefined behavior in the SPT environment.

Unless you are using the Standard POSIX Threads library, the **sa_flags** field can have the **SA_NOCLDSTOP** bit set to specify further control over the actions taken on delivery of a signal. If the *signal* parameter is **SIGCHLD** and a child process of the calling process stops, a **SIGCHLD** signal is sent to the calling process unless **SA_NOCLDSTOP** is set for **SIGCHLD**.

### Use From a Threaded Application

The thread-aware **sigaction( )** function allows the calling thread to change or examine the action to be taken on delivery of a specific signal. This call removes any previously established signal handler for this signal for this thread. You must reestablish the previous signal handler if you want to use it at a later time.

The thread-aware signal is always enabled in the POSIX User Thread Model library so that externally generated signals (such as **SIGINT**, **SIGQUIT**, **SIGALRM**, and **SIGCHLD**) are catchable by threads. When the thread library signal handler receives the signal, it will check to see if the current thread can handle the signal. If the current thread can handle the signal, then the thread library signal handler invokes the thread signal handler immediately. If the current thread cannot handle the signal, the thread library signal handler finds a thread that can handle the signal, adds the signal to the queue for that thread, and returns. The thread signal handlers for these queued signals are run either at thread dispatch or at the cancellation point

### Use With Standard POSIX Threads

When using standard POSIX threads, specify the **spthread.h** header file. The **signal.h** header file can be omitted.

A multi-threaded process can use the **sigaction( )** function to establish thread-specific actions for synchronous signals. Each thread can have its own signal handler routine.

When you use the standard POSIX threads version of **sigaction( )**:

- The **sigaction( )** function only modifies behavior for individual threads.

- The **sigaction( )** function only works for synchronous signals. Attempting to set a signal action for an asynchronous signal is an error. This is true even in a single-threaded process.
- The signal mask is manipulated using the following functions: **sigemptyset( )**, **sigfillset( )**, **sigaddset( )**, **sigdelset( )**, and **sigismember( )**.

For additional information on using the **sigaction( )** function in a threaded application that uses the Standard POSIX Threads library, see **Specifying Options**, earlier.

**NOTES**

A threaded application that uses the Standard POSIX Threads library may use the **spt_sigaction(2)** function instead of the **sigaction( )** function, however, for portability reasons, the use of **sigaction( )** is recommended.

To use the **spt_sigaction( )** function in a threaded application that uses the Standard POSIX Threads library, see **spt_sigaction(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion of the **sigaction( )** function, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

If the **SA_ONSTACK** flag and the **SA_ONSTACK_COMPATIBILITY** feature test macro are set, although the thread-aware **sigaction( )** function returns 0 (zero), the POSIX User Thread Model library determines which stack is used to run the signal handler.

## ERRORS

If any of the following conditions occurs, the **sigaction( )** function sets **errno** to the corresponding value and no new signal-catching function is installed:

[EFAULT] The *action* or *o_action* parameter points to a location outside of the allocated address space of the process.

[EINVAL] One of the following conditions exists:

- The *signal* parameter is not a valid signal number.
- An attempt was made to ignore or supply a signal-catching function for the **SIGKILL**, **SIGSTOP**, or **SIGABEND** signal.

[ENOTOSS] The calling process was not an OSS process or a native Guardian process. The requested operation cannot be performed from the Guardian environment by a TNS or accelerated Guardian process.

## RELATED INFORMATION

Commands: **kill(1)**.

Functions: **_exit(2)**, **exit(3)**, **kill(2)**, **pause(3)**, **pthread_sigmask(2)**, **setjmp(3)**, **sigaddset(3)**, **sigdelset(3)**, **sigemptyset(3)**, **sigfillset(3)**, **sigismember(3)**, **signal(3)**, **sigaltstack(2)**, **sigprocmask(2)**, **sigsuspend(2)**, **spt_sigaction(2) umask(2)**, **wait(2)**.

Files: **signal(4)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- The ordering of members within the **sigaction** structure might not match the ordering used in **signal.h** header files in other environments or on other systems.
- The values returned in the fields of the structure pointed to by the *o_action* parameter when **sigaction( )** is called and the previous action for the specified signal was established by the **signal( )** function are unspecified in the POSIX.1 standard. These values should therefore not be depended upon other than to pass the address returned in *o_action* as the *action* parameter to another **sigaction( )** function; the result is as if the **signal( )** function were repeated.
- The action is ignored when the action is set to the **SIG_DFL** value for a signal that cannot be caught or ignored.

This function is an extension to the XPG4 Version 2 specification.

The following are HP extensions to the XPG4 Version 2 specification:

- HP has defined several new signals, including **SIGABEND**. See the **signal(4)** reference page for a complete list.
- The [ENOTOSS] error value is an HP extension.
- If the **SIGSTK** signal is delivered while the alternate signal stack is active, the default action of terminating the process occurs.

HP does not define members of the **sigaction** structure following **sa_flags**.

HP does not define the **SA_SIGINFO** symbolic constant.

HP does not support the Realtime Signals Extension. The **errno** value [ENOTSUP] is not returned.

HP maintains only one alternate signal stack per process for unbound threads. If an alternate signal stack is registered, this alternate signal stack applies to all threads in the process. Note that this alternate signal stack behavior does not apply to bound threads.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

This function does not conform to the async-signal safe requirement of the POSIX.1 standard.

**NAME**

**sigaltstack -** Sets and gets the signal alternate stack context

**LIBRARY**

H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

**SYNOPSIS**

**#include <signal.h>**

**int sigaltstack(**
**const stack_t ****ss***,**
**stack_t ****oss***);**

**PARAMETERS**

*ss* Specifies the signal alternate stack context that is to be defined as a result of the current call to the **sigaltstack( )** function. If *ss* is a null pointer, no action is taken, but the current alternate signal stack state is returned in the **stack_t** structure pointed to by *oss*.

*oss* Points to a **stack_t** structure that returns the signal alternate stack context in effect before the call was made. If the parameter is not a null pointer, the alternate signal stack context in effect at the time of the **sigaltstack( )** call is returned.

**DESCRIPTION**

The **sigaltstack( )** function allows a process to define or examine the state of an alternate stack for signal handlers for the calling thread. Signals that have been explicitly declared to execute on the alternate stack are delivered on the alternate stack.

When the **sigaltstack( )** function is used in a threaded application that uses the POSIX User Thread Model library, this function only allows the user thread to examine the state of an alternate stack for signal handlers; it does not allow the user thread to define a new alternate stack. If **sigaltstack( )** is called to install a new alternate stack with this library, **sigaltstack( )** returns the value -1 and sets **errno** to [EINVAL].

**Use From the Guardian Environment**

If called from a TNS or accelerated Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

The **sigaltstack( )** function can be called from H-series or J-Series native Guardian processes on systems running J06.10 or later RVUs or H06.21 or later RVUs.

**Specifying Options**

If *ss* is not a null pointer, it points to a **stack_t** structure that specifies the alternate signal stack that takes effect upon return from **sigaltstack( )**. The *ss_flags* field specifies the new stack state. If it is set to **SS_DISABLE**, the stack is disabled and *ss_sp* and *ss_size* are ignored. Otherwise, the stack is enabled, and the *ss_sp* and *ss_size* fields specify the new address and size of the stack.

The **sigaltstack( )** function only accepts a stack pointer obtained from the **STACK_ALLOCATE_( )** procedure.

If *oss* is not a null pointer, on successful completion, it points to a **stack_t** structure that specifies the alternate signal stack that was in effect prior to the call to the **sigaltstack( )** function. The *ss_sp* and *ss_size* fields specify the address and size of that stack. The *ss_flags* field specifies the stack's state as one of the following values:

**SS_ONSTACK**
The process is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the process is executing on it fail. This flag cannot be modified by processes.

**SS_DISABLE** The alternate signal stack is currently disabled.

The value **SIGSTKSZ** is a system default that specifies the number of bytes that are usually required when manually allocating an alternate stack area. The value **MINSIGSTKSZ** is defined to be the minimum stack size for a signal handler. In computing an alternate stack size, an application should add the value **MINSIGSTKSZ** to its stack requirements to allow for the system implementation overhead. The constants **SS_ONSTACK**, **SS_DISABLE**, **SIGSTKSZ**, and **MINSIGSTKSZ** are defined in the **signal.h** header file.

After a successful call to one of the **exec** set of functions, no alternate signal stacks exist in the new process image. After a successful call to the **fork( )** function, the alternate signal stack exists in the child process at the same address and with the same contents.

A signal handler only runs on the alternate signal stack if the thread that defined the signal handler is not blocked when the signal is delivered. If the thread is blocked, the signal handler runs on the user stack.

The **SA_ONSTACK** flag (see **spt_sigaction(2)**) has no effect in the Guardian environment.

## NOTES

To ensure proper operation of the **fork( )** function, you must allocate alternate signal stacks as protected user stacks by setting the **ST_COF** (copy stack to child process upon **fork( )**) option of the **STACK_ALLOCATE_( )** procedure.

The **sigaltstack( )** requires the specified stack address and size describe exactly a user stack segment as created by the **STACK_ALLOCATE_( )** procedure. If the specified stack address and size do not describe a valid user stack segment as created by the **STACK_ALLOCATE_( )** procedure, **sigaltstack( )** returns the value -1 and sets **errno** to [EFAULT].

You should not use this function in a threaded application that uses the Standard POSIX Threads (SPT) library. Use of this function with the SPT library may result in undefined behavior.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

To call the **sigaltstack( )** function with the POSIX User Thread Model library, you must specify the *ss* parameter as a null pointer. The PUT library will automatically create an alternate signal stack on behalf of the process. You can use **sigaltstack( )** with the PUT library to obtain information about the alternate signal stack.

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion of the **sigaltstack( )** function, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **sigaltstack( )** function sets **errno** to the corresponding value and no action is taken:

[EFAULT] One of the following conditions exists:

- Either the *ss* parameter or the *oss* parameter references an invalid memory address.
- The specified stack address and size do not describe a valid user stack segment as created by the **STACK_ALLOCATE_( )** procedure.

[EINVAL] One of the following conditions exists:

- The *ss* parameter is not a null pointer and the *ss_flags* field pointed to by the *ss* parameter contains flags other than **SS_DISABLE**.
- The **sigaltstack( )** function is being used in a threaded application that uses the POSIX User Thread Model library and the *ss* parameter is not a null pointer.

[ENOMEM] The size of the alternate stack area is less than **MINSIGSTKSZ**.

[EPERM] An attempt was made to modify an active stack.

## RELATED INFORMATION

Commands: **kill(1)**.

Functions: **fork(2)**, **kill(2)**, **pause(3)**, **pthread_sigmask(2)**, **setjmp(3)**, **sigaction(2)**, **sigaddset(3)**, **sigdelset(3)**, **sigemptyset(3)**, **sigfillset(3)**, **sigismember(3)**, **signal(3)**, **sigprocmask(2)**, **sigsuspend(2)**, **umask(2)**, **wait(2)**.

Files: **signal(4)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- The ordering of fields within the **stack_t** structure might not match the ordering used in **signal.h** header files in other environments or on other systems.

The following are HP extensions to the IEEE Std 1003.1-2004, POSIX System Application Program Interface specification:

- HP has defined several new signals, including **SIGABEND**. See the **signal(4)** reference page for a complete list.
- If the **SIGSTK** signal is delivered while the alternate signal stack is active, the default action of terminating the process occurs.

## NAME

**sigpending -** Examines pending signals

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <signal.h>

int sigpending(
        sigset_t *set);
```

## PARAMETERS

*set* Points to an object of type **sigset_t** that returns the set of signals that are blocked from delivery and pending to the calling process.

## DESCRIPTION

The **sigpending( )** function stores the set of signals that are blocked from delivery and pending to the calling process in the object pointed to by the *set* parameter.

Because signals can arrive asynchronously, no assumption should be made about the current set of pending signals, based on the value returned by this function in *set*.

### Use From a Threaded Application

The thread-aware **sigpending( )** function retrieves the signals that have been sent to the calling thread but have been blocked from delivery. These signals are pending to the calling thread because the calling thread's signal mask is preventing their delivery. The blocked signals are stored in the structure pointed to by the *set* parameter.

### Use From the Guardian Environment

If called from a TNS or accelerated Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

## NOTES

To use the **sigpending( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_sigpending(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **sigpending( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **sigpending( )** function sets **errno** to the corresponding value:

[EFAULT] The *set* parameter points to a location outside the allocated address space of the process.

[ENOTOSS] The calling process was not an OSS process or a native Guardian process. The **sigpending( )** function cannot be used in the Guardian environment by a TNS or accelerated process.

**RELATED INFORMATION**

Functions: **sigaddset(2)**, **sigdelset(2)**, **sigemptyset(3)**, **sigfillset(2)**, **sigismember(2)**, **sigprocmask(2)**, **spt_sigpending(2)**.

Files: **signal(4)**.

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 Version 2 specification.

The following are HP extensions to the XPG4 Version 2 specification:

- The [EFAULT] and [ENOTOSS] errors can be returned.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

## NAME

**sigprocmask -** Changes or examines the signal mask

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

**#include <signal.h>**

**int sigprocmask(**
  **int** *how***,**
  **sigset_t ****set***,**
  **sigset_t ****o_set* **);**

## PARAMETERS

*how*
Indicates the manner in which the set of masked signals is changed; it has one of the following values:

**SIG_BLOCK** The resulting set is the union of the current set and the signal set pointed to by the *set* parameter.

**SIG_UNBLOCK**
The resulting set is the current set less the signals indicated in the signal set pointed to by the *set* parameter.

**SIG_SETMASK**
The resulting set is the signal set pointed to by the *set* parameter.

*set*
Specifies the signal set. If the *set* parameter is not a null pointer, it points to a set of signals to be used to change the currently blocked set. If the *set* parameter is a null pointer, the value of the *how* parameter is not significant and the process signal mask is unchanged; thus, the call can be used to inquire about currently blocked signals.

*o_set*
Returns the existing signal mask. If the *o_set* parameter is not a null pointer, the signal mask in effect at the time of the call is stored in the variable pointed to by the *o_set* parameter.

## DESCRIPTION

The **sigprocmask( )** function is used to change or examine the signal mask of the calling process.

Typical use is to

1. Call the **sigprocmask(SIG_BLOCK)** function to block signals during a critical section of code.

2. Call the **sigprocmask(SIG_SETMASK)** function at the end of the critical section of code to restore the mask to the previous value returned by the **sigprocmask(SIG_BLOCK)** function.

If there are any unblocked signals pending after a call to the **sigprocmask( )** function, at least one of those signals will be delivered before the **sigprocmask( )** function returns.

The **sigprocmask( )** function does not allow the **SIGKILL**, **SIGABEND**, or **SIGSTOP** signals to be blocked. If a program attempts to block any of these signals, the **sigprocmask( )** function gives no indication of the error.

Any signal that is generated by an event other than the **kill( )** or **raise( )** function causes process termination if the signal is blocked. If possible, a saveabend file is created.

### Use From the Guardian Environment

If called from a TNS or accelerated Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

## EXAMPLES

The following example shows how to use **sigprocmask(SIG_BLOCK)** to add the signal **SIGINT** to the signal set named `newset` and save the old mask. Later, the **sigprocmask(SIG_SETMASK)** function restores the mask to the previous value returned by the **sigprocmask(SIG_BLOCK)** function.

```
#include <signal.h>

int return_value;
sigset_t newset, oldset;

sigemptyset(&newset);
sigaddset(&newset, SIGINT);
return_value = sigprocmask (SIG_BLOCK, &newset, &oldset);
 . . .
return_value = sigprocmask (SIG_SETMASK, &oldset, NULL);
```

## RETURN VALUES

Upon successful completion, the **sigprocmask( )** function returns the value 0 (zero). If the **sigprocmask( )** function fails, the signal mask of the process is unchanged, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **sigprocmask( )** function sets **errno** to the corresponding value:

[EFAULT] The *set* or *o_set* parameter points to a location outside the allocated address space of the process.

[EINVAL] The value of the *how* parameter is not equal to one of the defined values.

[ENOTOSS] The calling process was not an OSS process or a native Guardian process. The **sigprocmask( )** function cannot be used in the Guardian environment by a TNS or accelerated Guardian process.

## RELATED INFORMATION

Functions: **kill(2)**, **pthread_sigmask(2)**, **sigaction(2)**, **sigaddset(2)**, **sigdelset(2)**, **sigemptyset(2)**, **sigfillset(2)**, **sigismember(2)**, **sigpending(2)**, **sigsuspend(2)**.

Files: **signal(4)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- HP has defined several new signals, including **SIGABEND**. See the **signal(4)** reference page for a complete list.
- This function can set **errno** to the value [ENOTOSS].

## NAME

**sigsuspend -** Changes the set of blocked signals and waits for a signal

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <signal.h>

int sigsuspend(
        sigset_t *signal_mask );
```

## PARAMETERS

*signal_mask* Points to a set of signals to be blocked from delivery to the calling process.

## DESCRIPTION

The **sigsuspend( )** function replaces the signal mask of the process with the set of signals pointed to by the *signal_mask* parameter, and then suspends execution of the process until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. The **sigsuspend( )** function does not allow the **SIGKILL**, **SIGABEND**, or **SIGSTOP** signals to be blocked. If a program attempts to block one of these signals, the **sigsuspend( )** function gives no indication of an error.

If delivery of a signal causes the process to terminate, the **sigsuspend( )** function does not return. If delivery of a signal causes a signal-catching function to execute, the **sigsuspend( )** function returns after the signal-catching function returns, with the signal mask restored to the set that existed prior to the call to the **sigsuspend( )** function.

The **sigsuspend( )** function sets the signal mask and waits for an unblocked signal as one atomic operation. This means that signals cannot occur between the operations of setting the mask and waiting for a signal.

In normal use, a signal is blocked by calling the **sigprocmask(SIG_BLOCK)** function at the beginning of a critical section of code. The process then determines whether there is work for it to do. If no work is to be done, the process waits for work by calling the **sigsuspend( )** function with the mask previously returned by the **sigprocmask( )** function.

### Use From a Threaded Application

The thread-aware **sigsuspend( )** function replaces the current signal mask of a thread with the signal set specified by the *signal_mask* parameter and suspends processing for the thread until the thread receives one of the following signals:

- **SIGSTOP**, **SIGKILL**, or **SIGABEND**.
- A signal that is not a member of *signal_mask* and has an action that either calls a signal-catching function, ends the request, or terminates the process.

The thread-aware signal is always enabled in the POSIX User Thread (PUT) library so that externally generated signals (such as **SIGINT**, **SIGQUIT**, **SIGALRM**, and **SIGCHLD**) are catchable by threads.

### Use From the Guardian Environment

If called from a TNS or accelerated Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

## NOTES

To use the **sigsuspend( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_sigsuspend(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

If a signal is caught by the calling process and control is returned from the signal-catching function, the calling process resumes execution after the **sigsuspend( )** function, which always returns the value -1 and, after finishing normally, sets **errno** to [EINTR].

## ERRORS

If any of the following conditions occur, the **sigsuspend( )** function sets **errno** to the corresponding value:

[EINTR] The **sigsuspend( )** function was interrupted by a signal that was caught by the calling process, and control was returned from the signal-catching function.

[ENOTOSS] The calling process was not an OSS process or a native Guardian process. This function cannot be used in the Guardian environment by a TNS or accelerated process.

If the *signal_mask* parameter points to an invalid location, the **sigsuspend( )** function generates an unspecified signal that cannot be blocked or ignored and sends the signal to the process.

## RELATED INFORMATION

Functions: **pause(3)**, **sigaction(2)**, **signal(3)**, **sigprocmask(2)**, **spt_sigsuspend(2)**.

Files: **signal(4)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification.

The following are HP extensions to the XPG4 Version 2 specification:

- HP has defined several new signals, including **SIGABEND**. See the **signal(4)** reference page for a complete list.
- The [ENOTOSS] **errno** value is an HP extension.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

This function does not conform to the async-signal safe requirement of the POSIX.1 standard.

## NAME

**sigwait -** Causes the calling thread to wait for a signal

## LIBRARY

32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <signal.h>

int sigwait(
        sigset_t *set
          int *sig);
```

## PARAMETERS

*set* Specifies the set of signals that the calling thread will wait for.

*sig* Receives the signal number cleared from the specified set of signal numbers.

## DESCRIPTION

This function causes a thread to wait for a signal. It atomically chooses a pending signal from the set of pending signals indicated by the *set* parameter, atomically clears that signal from the system's set of pending signals, and returns that signal number at the location specified by the *sig* parameter. If no signal in *set* is pending at the time of the call, the thread is blocked until one or more signals become pending. The signals defined by *set* should be unblocked during the call to this function and are blocked when the thread returns from the call.

A thread must block the signals it waits for using the **pthread_sigmask( )** function before calling this function.

If more than one thread is using this function to wait for the same signal, only one of those threads returns from this function with the signal number.

A call to the **sigwait( )** function is a cancellation point.

The thread-aware signal is always enabled in the POSIX User Thread (PUT) library so that externally generated signals (such as **SIGINT**, **SIGQUIT**, **SIGALRM**, and **SIGCHLD**) are catchable by threads.

## NOTES

The **sigwait( )** function is not supported for non-threaded applications.

To use the **sigwait( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_sigwait(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

HP recommends that you do not specify threads to wait for process-level signals such as **SIGCONT**, **SIGTTIN**, **SIGTTOU**, and **SIGTSTP**. If a thread uses a function such as thread-aware **sigwait( )** or thread-aware **sigsuspend( )**, the thread breaks from the wait state only if the corresponding signal is sent using the **pthread_kill( )** function. The thread does not break from the wait state for signals that are generated at the process level.

The **SIGCHLD** signal is delivered to the correct thread even though the **SIGCHLD** signal is generated asynchronously.

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

On a successful call, the signal number is returned; otherwise, [EINVAL] is returned.

## ERRORS

If the only signals passed are unsupported signals, the **sigwait( )** function returns an **errno** value of [EINVAL].

For the following signals, support by the **sigwait( )** function is dependent on the RVU runiing on the system:

- The **SIGUNCP** signal is supported on systems running H-series RVUs only. This signal is not supported on systems running J-series RVUs.
- The following signals are supported on systems running H06.06 or later RVUs:
  - **SIGCHLD**
  - **SIGCONT**
  - **SIGTSTP**
  - **SIGTTIN**
  - **SIGTTOU**

## RELATED INFORMATION

Functions: **pause(2)**, **pthread_cancel(2)**, **pthread_sigmask(2)**, **sigpending(2)**, **spt_sigwait(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

The HP implementation does not provide the **sigwaitinfo( )** or **sigtimedwait( )** functions.

**NAME**

**sockatmark -** Determines whether a socket is at the out-of-band mark

**LIBRARY**

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

**SYNOPSIS**

**#include <sys/socket.h>**

**int sockatmark(**
**int** *socket***);**

**PARAMETERS**

*socket* Specifies the file descriptor for the socket.

**DESCRIPTION**

The **sockatmark( )** function determines whether the specified socket is at an out-of-band mark in its receive queue data. Calls to the **sockatmark( )** function between receive operations allow an application to determine the position of out-of-band data within its received data.

A call to **sockatmark( )** does not remove the out-of-band data mark from the data stream.

**NOTES**

A call to the **sockatmark( )** function can be made instead of a call to the **ioctl( )** function with a request of **SIOCATMARK**.

**RETURN VALUES**

If the protocol has marked the data stream and all data preceeding the mark has been read, the **sockatmark( )** function returns the value 1. If no mark exists or if data precedes the mark in the receive queue, the function call returns the value 0 (zero).

If the **sockatmark( )** call fails, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **sockatmark( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid open file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time might succeed.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOTTY] The *socket* parameter does not refer to a socket.

## RELATED INFORMATION

Functions: **recv(2)**, **recvmsg(2)**, **socket(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 specification.

## NAME

**socket -** Creates an endpoint for communications

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <sys/socket.h>**

**int socket(**
  **int** *domain***,**
  **int** *type***,**
  **int** *protocol*
  **);**

## PARAMETERS

*domain* Specifies the address family of the communications domain in which the socket is to be created.

*type* Specifies the type of socket to be created.

*protocol* Specifies a particular protocol to be used with the created socket. Specifying a *protocol* of 0 (zero) causes the **socket( )** function to default to the typical protocol used for the requested socket type. If a nonzero value is specified for *protocol*, it must specify a protocol that is supported by the address family specified by the *domain* parameter.

## DESCRIPTION

The **socket( )** function creates an unbound socket in a specified communications domain and returns a file descriptor for the socket that can be used in later function calls that operate on sockets.

The *domain* parameter specifies the address family used in the communications domain. The address families supported are:

**AF_INET** IPv4 Internet addresses. The value **PF_INET** can also be used to specify this address family.

**AF_INET6** IPv6 Internet addresses. The value **PF_INET6** can also be used to specify this address family.

**AF_UNIX** UNIX pathnames. The values **PF_UNIX**, **AF_LOCAL**, and **PF_LOCAL** can also be used to specify this address family.

The *type* parameter specifies the socket type, which determines the semantics of communications over the socket. The socket types supported are:

**SOCK_DGRAM**
Provides datagrams, which are connectionless, unreliable messages of a fixed maximum length.

**SOCK_STREAM**
Provides sequenced, reliable, two-way, connection-oriented byte streams with a transmission mechanism for out-of-band data.

The documentation for specific address families specifies which socket types each family supports. The **sys/socket.h** header file contains definitions for socket domains, types, and protocols.

Socket-level options control socket operations. The **getsockopt( )** and **setsockopt( )** functions are used to get and set these options, which are defined in the **sys/socket.h** file.

### Use From the Guardian Environment

The **socket( )** function is one of a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.dtpa

### Choosing the Transport-Provider Process

Each socket declared by a user process is supported by:

- An OSS transport agent process (one per processor)
- A domain-specific transport-provider process (one or more per node)

Each user process has a current transport-provider name for each domain that is used when creating a socket in that domain.

### AF_INET or AFINET6 Sockets

The default **AF_INET** or **AF_INET6** transport-provider name is $ZTC0, unless overridden by an existing Guardian DEFINE =TCPIP^PROCESS^NAME. If =TCPIP^PROCESS^NAME exists, it must be a MAP DEFINE with a FILE attribute string of the desired **AF_INET** or **AF_INET6** transport-provider name.

Each user process can change its **AF_INET** or **AF_INET6** transport-provider name with the **socket_transport_name_set( )** function and can retrieve its current **AF_INET**, **AF_INET6**, and **AF_UNIX** transport-provider names with the **socket_transport_name_get( )** function.

Changing the **AF_INET** or **AF_INET6** transport-provider name is meaningful when a node is configured with multiple TCP/IP processes as part of the **AF_INET** or **AF_INET6** socket environment.

### AF_UNIX Sockets

If the *domain* is **AF_UNIX**, the **AF_UNIX** transport-provider name determines if the socket is an **AF_UNIX** Release 1 socket, an **AF_UNIX** Release 2 socket in compatibility mode, or an **AF_UNIX** Release 2 socket in portability mode:

$ZPLS For systems running **AF_UNIX** Release 1, $ZPLS is the only supported **AF_UNIX** transport-provider name and this name cannot be changed.

For systems running **AF_UNIX** Release 2 software, if the transport-provider name is $ZPLS, the socket is created as an **AF_UNIX** Release 2 socket in compatibility mode. Sockets in compatibility mode can communicate with each other but cannot communicate with sockets created in portability mode. The default **AF_UNIX** transport-provider name is $ZPLS,

$ZAFN2 This transport-provider name is only valid for systems running **AF_UNIX** Release 2 software. If the **AF_UNIX** transport-provider name is $ZAFN2, the socket is created as an **AF_UNIX** Release 2 socket in portability mode. Sockets created in portability mode can communicate with each other but cannot communicate with sockets created in compatibility mode.

For systems running **AF_UNIX** Release 2 software, the default **AF_UNIX** transport-provider name is $ZPLS, which results in sockets created in compatibility mode. You can set the transport-provider name using either the **socket_transport_name_set( )** function or the Guardian DEFINE =_AFUNIX_PROCESS_NAME. This Guardian define must be a MAP DEFINE with a FILE attribute string of the desired **AF_UNIX** transport-provider name.

For more information about **AF_UNIX** Release 2 sockets, portability mode, and compatibility mode, see the *Open System Services Programmer's Guide*.

### Preventing Memory Conflicts

OSS socket applications use QIO shared memory to exchange data with the OSS transport agent. QIO uses some areas of memory for internal message processing, and it is necessary to prevent OSS socket applications from using overlapping memory ranges. Do either of these things to prevent memory conflicts between your socket application and QIO:

- If possible, do not specify a flat segment address; instead allow the operating system to allocate a starting region for you by specifying a null pointer as the second parameter in the call to the **shmat( )** function, as shown:

```
char *base_Ptr = shmat(shmid, (void*) 0, shmflag);
```

- If you do assign a base address, do not use any address in the range `0x20000000` through `0x41FFFFFF` or any of these flat segment regions:

```
0x20000000  0x22000000  0x24000000  0x26000000
0x28000000  0x2A000000  0x2C000000  0x2E000000
0x30000000  0x32000000  0x34000000  0x36000000
0x38000000  0x3A000000  0x3C000000  0x3E000000
0x40000000  0x42000000  0x44000000  0x46000000
0x48000000  0x4A000000  0x4C000000  0x4E000000
```

On some processors, QIO allocates its shared memory region by default starting at `0x20000000`, so a QIO configuration of 544 megabytes uses the flat segment regions listed previously. If you use a null pointer instead of specifying the base address, the operating system allocates flat segments for your application starting from the topmost region downward. As a result, OSS socket applications can safely allocate flat segments in any of the upper regions of memory on that processor.

If a memory conflict error still occurs, either you must change the memory allocation for your application or your system administrator must reconfigure QIO.

For more information on the **shmat( )** function, see the **shmat(2)** reference page. For more information on memory addressing, see the *Guardian Programmer's Guide* and the server description manual appropriate for your system.

## RETURN VALUES

Upon successful completion, the **socket( )** function returns the file descriptor for the socket. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **socket( )** function sets **errno** to the corresponding value:

[EACCES] The process does not have appropriate privileges to create the socket.

[EAFNOSUPPORT]
The specified address family is not supported.

[EDEFINEERR]
The Guardian DEFINE =TCPIP^PROCESS^NAME is invalid, and the DEFINE was used in an attempt to set the transport-provider name.

[EMFILE] No more file descriptors are available for this process.

[ENFILE] One of these conditions exists:

- The maximum number of file descriptors of this file type (socket, pipe, etc.) for this processor are already open.
- The limit for open file descriptors of this file type has not been exceeded, but the maximum number of all file descriptors for this processor are already open.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time might succeed.

[ENOENT] One of these conditions occurred:

- The transport-provider process terminated abnormally.
- The transport-provider process has not been started.
- A previous call to the **socket_transport_name_set( )** function specified a nonexistent transport-provider process.
- The system is running **AF_UNIX** Release 1 and the OSS sockets local server process is not running.
- The system is running **AF_UNIX** Release 2 and processor in which the calling process is running either:
  - Does not have an OSS sockets local server 2 process that is running, or
  - Does not have a transport-agent process that is running.

  One of these configuration errors might have occurred:

  - A requested transport provider process is not available.
  - An initialization error occurred because of a QIO address conflict.
  - An initialization error occurred because of a parallel TCP/IP address conflict.

  The name of an unavailable transport-provider process can be obtained by a call to the **socket_transport_name_get( )** function. You can set the transport-provider name using either the **socket_transport_name_set( )** function or the Guardian DEFINE =_AFUNIX_PROCESS_NAME. This Guardian define must be a MAP DEFINE with a FILE attribute string of the desired **AF_UNIX** transport-provider name.

[ENOMEM] There was insufficient memory available to complete the operation.

[EPROTONOSUPPORT]
The specified address family does not support the specified protocol.

[EPROTOTYPE]
The specified socket type is not supported by the protocol.

[ETANOTRUNNING]
The OSS transport agent for this processor is not running.

This error also can occur when the calling process has migrated to a new processor that does not have a transport agent to support sockets. The socket can only be closed.

## RELATED INFORMATION

Functions: **accept(2)**, **bind(2)**, **connect(2)**, **getsockname(2)**, **getsockopt(2)**, **listen(2)**, **recv(2)**, **recvfrom(2)**, **recvmsg(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **setsockopt(2)**, **shmat(2)**, **shutdown(2)**, **socketpair(2)**, **socket_transport_name_get(2)**, **socket_transport_name_set(2)**.

## STANDARDS CONFORMANCE

The HP implementation does not return the **errno** value [ENOSR].

The HP implementation does not support the **SOCK_SEQPACKET** socket type.

HP extensions to the XPG4 specification are:

- The **errno** values [EDEFINEERR], [ENOENT], and [ETANOTRUNNING] can be returned.

## NAME

**socketpair -** Creates a pair of connected sockets

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <sys/socket.h>**

**int socketpair(**
    **int** *domain***,**
    **int** *type***,**
    **int** *protocol***,**
    **int** *socket_vector***[2]**
    **);**

## PARAMETERS

*domain* Specifies the communications domain in which the sockets are created. This parameter must be set to **AF_UNIX**.

*type* Specifies the type of sockets to create.

*protocol* Specifies the communications protocol that the socket pair will use. Specifying a value of 0 (zero) for this parameter causes the **socketpair( )** function to default to the typical protocol used for the requested socket type.

*socket_vector* Specifies a 2-integer array used to hold the file descriptors of the socket pair created with this function call.

## DESCRIPTION

The **socketpair( )** function creates an unbound pair of connected sockets in the domain specified by the *domain* parameter, of the type specified by the *type* parameter, under the protocol optionally specified by the *protocol* parameter.

The two sockets created by the **socketpair( )** function are identical. The file descriptors for the socket pair are returned in *socket_vector***[0]** and *socket_vector***[1]**.

The *domain* parameter specifies the address family used in the communications domain. The **socketpair( )** function supports only the **AF_UNIX** address family, which supports the use of UNIX pathnames.

The *type* parameter specifies the socket type, which determines the communication semantics that the socket pair will use. The socket types supported are:

**SOCK_DGRAM**
Provides datagrams, which are connectionless, unreliable messages of a fixed maximum length.

**SOCK_STREAM**
Provides sequenced, reliable, two-way, connection-oriented byte streams with a transmission mechanism for out-of-band data.

The documentation for specific address families specifies which socket types each family supports. The **sys/socket.h** header file contains definitions for socket domains, types, and protocols.

Socket-level options control socket operations. The **getsockopt( )** and **setsockopt( )** functions are used to get and set these options, which are defined in the **sys/socket.h** file.

If the *domain* is **AF_UNIX**, the **AF_UNIX** transport-provider name determines if the socket is an **AF_UNIX** Release 1 socket, an **AF_UNIX** Release 2 socket in compatibility mode, or an **AF_UNIX** Release 2 socket in portability mode:

$ZPLS For systems running **AF_UNIX** Release 1, $ZPLS is the only supported **AF_UNIX** transport-provider name and this name cannot be changed.

For systems running **AF_UNIX** Release 2 software, If the transport-provider name is $ZPLS, the socket is created as an **AF_UNIX** Release 2 socket in compatibility mode. Sockets in compatibility mode can communicate with each other but cannot communicate with sockets created in portability mode. The default **AF_UNIX** transport-provider name is $ZPLS,

$ZAFN2 This transport-provider name is only valid for systems running **AF_UNIX** Release 2 software. If the **AF_UNIX** transport-provider name is $ZAFN2, the socket is created as an **AF_UNIX** Release 2 socket in portability mode. Sockets created in portability mode can communicate with each other but cannot communicate with sockets created in compatibility mode.

For systems running **AF_UNIX** Release 2 software, the default **AF_UNIX** transport-provider name is $ZPLS, which results in sockets created in compatibility mode. You can set the transport-provider name using either the **socket_transport_name_set( )** function or the Guardian DEFINE =_AFUNIX_PROCESS_NAME. This Guardian define must be a MAP DEFINE with a FILE attribute string of the desired **AF_UNIX** transport-provider name.

For more information about **AF_UNIX** Release 2 sockets, portability mode, and compatibility mode, see the *Open System Services Programmer's Guide*.

### Use From the Guardian Environment

The **socketpair( )** function is one of a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

### Preventing Memory Conflicts

OSS socket applications use QIO shared memory to exchange data with the OSS transport agent. QIO uses some areas of memory for internal message processing, and it is necessary to prevent OSS socket applications from using overlapping memory ranges. Do either of the following things to prevent memory conflicts between your socket application and QIO:

- If possible, do not specify a flat segment address; instead allow the operating system to allocate a starting region for you by specifying a null pointer as the second parameter in the call to the **shmat( )** function, as shown:

```
char *base_Ptr = shmat(shmid, (void*) 0, shmflag);
```

- If you do assign a base address, do not use any address in the range `0x20000000` through `0x41FFFFFF` or any of these flat segment regions:

```
0x20000000  0x22000000  0x24000000  0x26000000
0x28000000  0x2A000000  0x2C000000  0x2E000000
0x30000000  0x32000000  0x34000000  0x36000000
0x38000000  0x3A000000  0x3C000000  0x3E000000
0x40000000  0x42000000  0x44000000  0x46000000
0x48000000  0x4A000000  0x4C000000  0x4E000000
```

On some processors, QIO allocates its shared memory region by default starting at `0x20000000`, so a QIO configuration of 544 megabytes uses the flat segment regions listed previously. If you use a null pointer instead of specifying the base address, the operating system allocates flat segments for your application starting from the topmost region downward. As a result, OSS socket applications can safely allocate flat segments in any of the upper regions of memory on that processor.

If a memory conflict error still occurs, either you must change the memory allocation for your application or your system administrator must reconfigure QIO.

For more information on the **shmat( )** function, see the **shmat(2)** reference page. For more information on memory addressing, see the *Guardian Programmer's Guide* and the server description manual appropriate for your system.

**RETURN VALUES**

Upon successful completion, the **socketpair( )** function returns the value 0 (zero). Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **socketpair( )** function sets **errno** to the corresponding value:

[EACCES] The process does not have appropriate privileges to create a socket.

[EAFNOSUPPORT]
The specified address family is not supported.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EMFILE] No more file descriptors are available for this process.

[ENFILE] One of these conditions exists:

- The maximum number of file descriptors of this file type (socket, pipe, etc.) for this processor are already open.
- The limit for open file descriptors of this file type has not been exceeded, but the maximum number of all file descriptors for this processor are already open.

[ENOBUFS] There was not enough buffer space available to complete the call. A retry at a later time might succeed.

[ENOENT] One of these conditions occurred:

- The transport-provider process terminated abnormally.
- The transport-provider process has not been started.

- A previous call to the **socket_transport_name_set( )** function specified a nonexistent transport-provider process.
- The system is running **AF_UNIX** Release 1 and the OSS sockets local server process is not running.
- The system is running **AF_UNIX** Release 2 and processor in which the calling process is running either:
  - Does not have an OSS sockets local server 2 process that is running, or
  - Does not have a transport-agent process that is running.

  One of these configuration errors might have occurred:
  - A requested transport provider process is not available.
  - An initialization error occurred because of a QIO address conflict.
  - An initialization error occurred because of a parallel TCP/IP address conflict.

  The name of an unavailable transport-provider process can be obtained by a call to the **socket_transport_name_get( )** function. You can set the transport-provider name using either the **socket_transport_name_set( )** function or the Guardian DEFINE =_AFUNIX_PROCESS_NAME. This Guardian define must be a MAP DEFINE with a FILE attribute string of the desired **AF_UNIX** transport-provider name.

[ENOMEM] There was insufficient memory available to complete the operation.

[EOPNOTSUPP]
The specified protocol does not permit the creation of socket pairs.

[EPROTONOSUPPORT]
The specified address family does not support the specified protocol.

[EPROTOTYPE]
The specified socket type is not supported by the protocol.

[ETANOTRUNNING]
The OSS transport agent for this processor is not running.

This error can also occur when the calling process has migrated to a new processor that does not have a transport agent to support sockets. The socket can only be closed.

## RELATED INFORMATION

Functions: **accept(2)**, **bind(2)**, **connect(2)**, **getsockname(2)**, **getsockopt(2)**, **listen(2)**, **recv(2)**, **recvfrom(2)**, **recvmsg(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **setsockopt(2)**, **shutdown(2)**, **socket(2)**, **socket_transport_name_get(2)**, **socket_transport_name_set(2)**.

## STANDARDS CONFORMANCE

The HP implementation does not support the **SOCK_SEQPACKET** socket type.

The HP implementation does not return the **errno** value [ENOSR].

HP extensions to the XPG4 specification are:

- The **errno** values [ENOENT] and [ETANOTRUNNING] can be returned.

## NAME

**socket_transport_name_get -** Gets the name of the transport-provider process

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <sys/socket.h>**

**int socket_transport_name_get(**
 **int** *domain***,**
 **char *** *buffer***,**
 **int** *maxlen*
 **);**

## PARAMETERS

*domain* Specifies the domain for which the transport-provider process name should be obtained. The following values are valid:

**AF_INET** Specifies the Internet domain using IPv4 addresses

**AF_INET6** Specifies the Internet domain using IPv6 addresses

**AF_UNIX** Specifies the local sockets domain

*buffer* Points to the buffer to contain the null-terminated transport-provider process name.

*maxlen* Specifies the length in bytes of the buffer pointed to by the *buffer* parameter. This value should be at least 9 so that the buffer is large enough to contain the null terminator and an 8-character process name.

## DESCRIPTION

The **socket_transport_name_get( )** function returns the name of the transport-provider process for the indicated domain as set by the most recent call to the **socket_transport_name_set( )** function, or it returns the default value if no calls to the **socket_transport_name_set( )** function have been made. The default transport-provider processes for each domain are as follows:

**AF_INET** or **AF_INET6**
The default transport-provider process is $ZTC0, unless overridden by an existing Guardian DEFINE =TCPIP^PROCESS^NAME.

**AF_UNIX** For systems running **AF_UNIX** Release 1 software, $ZPLS is the only supported **AF_UNIX** transport-provider name. The default transport-provider name is $ZPLS.

For systems running **AF_UNIX** Release 2 software:

The transport-provider name $ZAFN2 indicates that this is an **AF_UNIX** Release 2 socket in portability mode.

The transport-provider name $ZPLS indicates that this is an **AF_UNIX** Release 2 socket in compatibility mode. The default transport-provider name is $ZPLS.

For more information about **AF_UNIX** Release 2, see the *Open System Services Programmer's Guide***.**

The value returned in the buffer pointed to by the *buffer* parameter is always an uppercase name.

**NOTES**

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

**Choosing the Transport-Provider Process**

Each socket declared by a user process is supported by:

- An OSS transport agent process (one per processor)
- A domain-specific transport-provider process (one or more per node)

Each user process has a current transport-provider name for each domain that is used when creating a socket in that domain.

The default **AF_INET** or **AF_INET6** transport-provider name is $ZTC0, unless overridden by an existing Guardian DEFINE =TCPIP^PROCESS^NAME. If =TCPIP^PROCESS^NAME exists, it must be a MAP DEFINE with a FILE attribute string of the desired **AF_INET** or **AF_INET6** transport-provider name.

For systems running **AF_UNIX** Release 1 software, $ZPLS is the only supported **AF_UNIX** transport-provider name and this name cannot be changed.

For systems running **AF_UNIX** Release 2 software:

- The default **AF_UNIX** transport-provider name is $ZPLS, unless it has been overridden by the setting DEFINE =_AFUNIX_PROCESS_NAME to $ZAFN2. This define must be a MAP DEFINE with a FILE attribute string of the desired **AF_UNIX** transport-provider name.
- To create a socket in portability mode, choose transport-provider name $ZAFN2. Sockets created in portability mode can communicate with each other but cannot communicate with sockets created in compatibility mode.
- To create a socket in compatibility mode, choose transport-provider name $ZPLS. Sockets created in compatibility mode can communicate with each other but cannot communicate with sockets created in portability mode.

For more information about **AF_UNIX** Release 2, see the *Open System Services Programmer's Guide*.

Each user process can change its **AF_INET**, **AF_INET6**, or **AF_UNIX** transport-provider name with the **socket_transport_name_set( )** function and can retrieve its current **AF_INET**, **AF_INET6**, and **AF_UNIX** transport-provider names with the **socket_transport_name_get( )** function. For systems running the **AF_UNIX** Release 2 software, you can change the transport-provider name using either the **socket_transport_name_set( )** function or the Guardian DEFINE =_AFUNIX_PROCESS_NAME. This define must be a MAP DEFINE with a FILE attribute string of the desired **AF_UNIX** transport-provider name.

For more information about **AF_UNIX** Release 2, see the *Open System Services Programmer's Guide*.

Changing the **AF_INET** or **AF_INET6** transport-provider name is meaningful when a node is configured with multiple TCP/IP processes as part of the **AF_INET** or **AF_INET6** socket environment.

The transport-provider name is a convention and does not guarantee use of a specific TCP/IP stack. For example, on older systems, $ZTC0 provided only Internet Protocol version 4 addressing for an **AF_INET** stack and could be used to distinguish the stack to use for sockets that do not use **AF_INET6** features. On current systems, $ZTC0 might identify an **AF_INET6** protocol stack; check with your TCP/IP administrator to determine your site's naming conventions before using this function to distinguish between stacks.

**RETURN VALUES**

Upon successful completion, the **socket_transport_name_get( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **socket_transport_name_get( )** function sets **errno** to the corresponding value:

[EDEFINEERR]
One of the following conditions occurred:

- The Guardian DEFINE =TCPIP^PROCESS^NAME is invalid.
- The Guardian DEFINE =_AFUNIX_PROCESS_NAME is invalid.

[EFAULT] The address specified for the *buffer* parameter is not valid.

[EINVAL] One of the following conditions occurred:

- The *domain* parameter does not specify a supported domain.
- The buffer specified by the *buffer* and *maxlen* parameters is too small to hold the transport-provider process name.

**RELATED INFORMATION**

Functions: **socket_transport_name_set(2)**, **socket(2)**, **socketpair(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 specification.

## NAME

**socket_transport_name_set -** Sets the name of the transport-provider process

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <sys/socket.h>**

**int socket_transport_name_set(**
**int** *domain***,**
**char ****buffer*
**);**

## PARAMETERS

*domain* Specifies the domain for which the transport-provider process name is being set. The following values are valid:

**AF_INET** Specifies the Internet domain using IPv4 addresses

**AF_INET6** Specifies the Internet domain using IPv6 addresses

**AF_UNIX** Specifies the local sockets domain

*buffer* Points to the buffer that contains the null-terminated transport-provider process name. The buffer should be at most 9 characters long, to contain an 8-character process name and a null terminator. The name can be specified in lowercase letters; the name is always stored in uppercase characters.

## DESCRIPTION

The **socket_transport_name_set( )** function sets the name of the transport-provider process for the domain specified by the *domain* parameter. A subsequent call to the **socket_transport_name_get( )** function can obtain the value set by this function.

Standard socket behavior does not require use of this function. A default transport-provider process always exists for each domain that provides sockets, as follows:

**AF_INET** or **AF_INET6**
The default transport-provider process is $ZTC0, unless overridden by an existing Guardian DEFINE =TCPIPˆPROCESSˆNAME.

**AF_UNIX** For systems running **AF_UNIX** Release 1 software, $ZPLS is the only supported **AF_UNIX** transport-provider name. The default transport-provider name is $ZPLS.

For systems running **AF_UNIX** Release 2 software:

Choose transport-provider name $ZAFN2 to create an **AF_UNIX** Release 2 socket in portability mode. To set $ZAFN2 as the default transport-provider name, you can use the Guardian DEFINE =_AFUNIX_PROCESS_NAME and specify $ZAFN2. This Guardian define must be a MAP DEFINE with a FILE attribute string of the desired AF_UNIX transport provider name.

Choose transport-provider name $ZPLS to create an **AF_UNIX** Release 2 socket in compatibility mode. The default transport-provider name is $ZPLS.

For more information about **AF_UNIX** Release 2, see the *Open System Services Programmer's Guide***.**

## NOTES

This function is equivalent to the **socket_set_inet_name( )** function in the Guardian sockets library.

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

The process name specified in the **socket_transport_name_set( )** function call is validated during each call to the **socket()**, **socketpair( )**, or **socket_transport_name_get( )** function. Process names are not validated during calls to the **socket_transport_name_set( )** function.

### Choosing the Transport-Provider Process

Each socket declared by a user process is supported by:

- An OSS transport agent process (one per processor)
- A domain-specific transport-provider process (one or more per node)

Each user process has a current transport-provider name for each domain that is used when creating a socket in that domain.

The default **AF_INET** or **AF_INET6** transport-provider name is $ZTC0, unless overridden by an existing Guardian DEFINE =TCPIP^PROCESS^NAME. If =TCPIP^PROCESS^NAME exists, it must be a MAP DEFINE with a FILE attribute string of the desired **AF_INET** or **AF_INET6** transport-provider name.

For systems running **AF_UNIX** Release 1 software, $ZPLS is the only supported **AF_UNIX** transport-provider name and this name cannot be changed.

For systems running **AF_UNIX** Release 2 software:

- The default **AF_UNIX** transport-provider name is $ZPLS, unless it has been overridden by the setting the Guardian DEFINE =_AFUNIX_PROCESS_NAME to $ZAFN2. This define must be a MAP DEFINE with a FILE attribute string of the desired **AF_UNIX** transport-provider name.
- To create a socket in portability mode, choose transport-provider name $ZAFN2. Sockets created in portability mode can communicate with each other but cannot communicate with sockets created in compatibility mode.
- To create a socket in compatibility mode, choose transport-provider name $ZPLS. Sockets created in compatibility mode can communicate with each other but cannot communicate with sockets created in portability mode.

For more information about AF_UNIX Release 2, see the *Open System Services Programmer's Guide***.**

Each user process can change its **AF_INET**, **AF_INET6**, or **AF_UNIX** transport-provider name with the **socket_transport_name_set( )** function and can retrieve its current **AF_INET**, **AF_INET6**, and **AF_UNIX** transport-provider names with the **socket_transport_name_get( )** function.

Changing the **AF_INET** or **AF_INET6** transport-provider name is meaningful when a node is configured with multiple TCP/IP processes as part of the **AF_INET** or **AF_INET6** socket environment or the Cluster I/O Protocols (CIP) networking environment. When using the CIP networking environment, you must choose a transport provider that is a CIPSAM process. The default transport-provider name is $ZTC0. The default program name is CIPSAM. For more information about the CIP networking environment, see the *Cluster I/O Protocols (CIP) Configuration and Management Manual*.

**RETURN VALUES**

Upon successful completion, the **socket_transport_name_set( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the **socket_transport_name_set( )** function sets **errno** to the corresponding value:

[EFAULT] The address specified for the *buffer* parameter is not valid.

[EINVAL] One of the following conditions occurred:

- The *domain* parameter does not specify a supported domain.
- The null-terminated process name pointed to by the *buffer* parameter has zero length or is too large for a valid process name.

**RELATED INFORMATION**

Functions: **socket_transport_name_get(2)**, **socket(2)**, **socketpair(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 specification.

## NAME

**spt_accept -** Initiates thread-aware **accept( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <spthread.h>**

**int spt_accept(**
    **int** *socket***,**
    **struct sockaddr ****address***,**
    **size_t ****address_len***);**

## PARAMETERS

See the **accept(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **accept( )** function. The socket must be nonblocking for this function to be thread aware.

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

The following macro maps **spt_accept( )** to **accept( )** and has been defined in **spthread.h**:

```
#define accept(socket, address, address_len) \
        spt_accept(socket, address, address_len)
```

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE
```

## RETURN VALUES

See the **accept(2)** reference page. The following also applies:

- The returned file descriptor is nonblocking.
- Value **errno** is never set to [EWOULDBLOCK].
- If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** of [EBADF].
- If a signal is received via **pthread_kill( )** and is not blocked, ignored, or handled, -1 is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_acceptx -** Accepts a new connection on a socket (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <sys/socket.h>**]
**#include <spthread.h>**

**int spt_acceptx(**
    **int** *socket***,**
    **struct sockaddr ***address***,**
    **size_t ***address_len*
    **);**

## PARAMETERS

*socket* Specifies the file descriptor for a socket that was created with the **socket( )** function, has been bound to an address with the **bind( )** function, and has issued a successful call to the **listen( )** function.

*address* Specifies either a null pointer or a pointer to the **sockaddr** structure where the address of the peer socket that requested the connection should be returned. The length and format of the address depend on the address family of the socket.

For **AF_INET** sockets, a pointer to the address structure **sockaddr_in** must be cast as a **struct sockaddr**. For **AF_INET6** sockets, a pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**. For **AF_UNIX** sockets, a pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*address_len* Points to a **size_t** data item, which, on input, specifies the length of the **sockaddr** structure pointed to by the *address* parameter, and, on output, specifies the length of the address returned.

## DESCRIPTION

The **spt_acceptx( )** function is a thread-aware version of the **accept( )** function.

The **spt_acceptx( )** function extracts the first connection on the queue of pending connections, creates a new socket with the same socket type, protocol, and address family as the specified socket, and allocates a new file descriptor for that socket.

When the **spt_acceptx( )** function is called using a value for the *address* parameter that is null, successful completion of the call returns a socket file descriptor without modifying the value pointed to by the *address_len* parameter. When the **spt_acceptx( )** function is called using a value for the *address* parameter that is not null, a successful call places the address of the peer socket in the **sockaddr** structure pointed to by the *address* parameter, and places the length of the peer socket's address in the location pointed to by the *address_len* parameter.

If the length of the socket address is greater than the length of the supplied **sockaddr** structure, the address is truncated when stored.

If the queue of pending connections is empty of connection requests and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **spt_acceptx( )** function blocks until a connection is present. If the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the queue of pending connections is empty, the **spt_acceptx( )** function call fails and sets **errno** to [EWOULDBLOCK].

## NOTES

The macro to map **accept( )** to **spt_acceptx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **accept( )** to **spt_acceptx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

When a connection is available, a call to the **select( )** function indicates that the file descriptor for the original socket is ready for reading.

The accepted socket cannot itself accept more connections. The original socket remains open and can accept more connections.

## RETURN VALUES

Upon successful completion, the **spt_acceptx( )** function returns the file descriptor of the accepted socket. If the **spt_acceptx( )** function call fails, the value -1 is returned and **errno** is set to indicate the error.

If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## ERRORS

If any of the following conditions occurs, the **spt_acceptx( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNABORTED]
The connection was aborted.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINTR] The function call was interrupted by a signal that was caught before a valid connection arrived.

[EINVAL] The socket is not accepting connections.

[EMFILE] No more file descriptors are available for this process.

[ENFILE] The maximum number of file descriptors for this processor are already open.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOTSOCK] The *socket* parameter does not specify a socket.

[EOPNOTSUPP]
The socket type of the specified socket does not support accepting connections.

[EWOULDBLOCK]
The socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set) and no connections are present to be accepted.

## RELATED INFORMATION

Functions: **accept(2)**, **bind(2)**, **connect(2)**, **fcntl(2)**, **listen(2)**, **socket(2)**, **spt_accept(2)**, **pthread_kill(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

The XPG4 specification allows certain behaviors of **accept( )** to be implementer-defined. For an indication of the HP implementation behaviors, see the **accept(2)** reference page either online or in the *Open System Services System Calls Reference Manual*.

## NAME

**spt_alarm** - Schedules an alarm signal for delivery to a process (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**unsigned int spt_alarm(**
**nsigned int ****seconds***);**

## PARAMETERS

*seconds* Specifies the number of real-time seconds to wait before sending the SIGALRM signal.

## DESCRIPTION

The **spt_alarm( )** function instructs the alarm clock of the calling thread to send the signal **SIGALRM** to a process after the number of real-time seconds specified by *seconds* has elapsed. If the value of *seconds* is 0 (zero), any previously set alarm is cancelled and no new alarm is scheduled. Each call to **spt_alarm( )** from a thread supersedes any previous calls from the same thread. If the same thread calls the **spt_alarm( )** function multiple times, the value of the *seconds* parameter from the most recent call is used.

To enable thread-aware behavior for this function, you must export the SPT_THREAD_AWARE_SIGNAL environmental variable to the value 1. By default, SPT_THREAD_AWARE_SIGNAL is disabled. If you do not export SPT_THREAD_AWARE_SIGNAL to 1, the **spt_alarm( )** function behaves as a process-level alarm (see the **alarm(3)** reference page).

In **spthread.h**, a mapping of **alarm( )** to **spt_alarm( )** has been defined:

**#define alarm(***seconds***) spt_alarm(***seconds***)**

For C applications, this mapping is available only when you define the correct preprocessor before you include **spthread.h**:

**#define SPT_THREAD_SIGNAL**
**#include <spthread.h>**

For C++ applications, this mapping is available only when you define the correct preprocessor before you include **spthread.h**:

**#define SPT_THREAD_SIGNAL_PRAGMA**
**#include <spthread.h>**

## NOTES

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

If a previous call to the **spt_alarm(2)** function has time remaining, this call to the **spt_alarm(2)** function returns the number of seconds remaining. Otherwise this function returns a value of 0 (zero).

## RELATED INFORMATION

Functions: **alarm(3)**, **pthread_kill(2)**, **pthread_sigmask(2)**, **spt_pause(2)**, **spt_sigaction(2)**, **spt_signal(2)**, **spt_sigsuspend(2)**, **spt_sigwait(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_awaitio -** Awaits a tagged I/O file

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**spt_error_t spt_awaitio(**
  **const short** *filenum***,**
  **const long** *tag***,**
  **const long** *timelimit***,**
  **long ****count_transferred***,**
  **long ****error***,**
  **void ****userdata***);**

## PARAMETERS

*filenum* Specifies Guardian file number being waited on

*tag* Specifies tag being waited on

*timelimit* Specifies how many hundredths of a second to wait for a completed I/O:

| | |
|---|---|
| -1 | means wait forever |
| 0 | means immediate return |

*count_transferred*
Specifies transfer count of completed I/O; set by callback when **SPT_SUCCESS** is returned.

*error* Specifies Guardian error number for I/O; set by callback when **SPT_SUCCESS** is returned or as described in **ERRORS**

*userdata* Specifies address of user data area; the referenced data may be modified by a callback

## DESCRIPTION

Awaits a tagged I/O on file number to complete, timeout, or be interrupted (see the **spt_interrupt(2)** reference page under **RETURN VALUES**). The function never cancels I/O. I/O completes only if **SPT_SUCCESS** is returned. Multiple threads should not await the same tagged I/O on any given file number.

## RETURN VALUES

**SPT_SUCCESS**
File number was waited on.

**SPT_ERROR** An error occurred. See **ERRORS**.

**SPT_TIMEDOUT**
Time limit has expired. See **ERRORS**.

**SPT_INTERRUPTED**
Wait was interrupted. See **ERRORS**.

## ERRORS

16 *filenum* is not registered.

29 *filenum* < 0 (zero).

40 *timelimit* has expired.

[EINTR] Wait was interrupted via **spt_interrupt( )**, **spt_interruptTag( )**, or a signal was received via **pthread_kill( )** and is not blocked, ignored, or handled.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_CANCEL -** Cancels the oldest incomplete operation on a Guardian file opened for nowait I/O

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_CANCEL (**
    **short** *filenum***);**

## PARAMETERS

*filenum* specifies the Guardian file number of a Guardian file open instance whose oldest incomplete operation you want to cancel

## DESCRIPTION

The **SPT_CANCEL( )** function is the thread-aware version of the Guardian CANCEL procedure.

The **SPT_CANCEL( )** function is used to cancel the oldest incomplete operation on a Guardian file opened for nowait I/O. The canceled operation might or might not have had effects. For disk files, the file position might or might not be changed.

For programming information about the Guardian CANCEL procedure, see the *Guardian Programmer's Guide*.

### Considerations

Queue files If an **SPT_READUPDATELOCKX( )** function operation is canceled using the **SPT_CANCEL( )** function, the **SPT_READUPDATELOCKX( )** call might already have deleted a record from the queue file, which could result in the loss of a record from the queue file. For audited queue files only, your application can recover from a timeout error by calling the **SPT_ABORTTRANSACTION( )** function, when detecting Guardian file-system error 40, to ensure that any dequeued records are reinserted into the file.

For nonaudited queue files, there is no recovery of a lost record. Thus, your application should never call the Guardian AWAITIOX procedure with a time limit greater than 0 (zero) if an **SPT_READUPDATELOCKX( )** call is pending. The **SPT_ABORTTRANSACTION( )** recovery procedure does not work on nonaudited queue files.

Messages The server process (that is, a process that was opened and to which the I/O request was sent) receives a system message -38 (queued message cancellation) that identifies the canceled I/O request, if it has requested receipt of such messages. If the server has already replied to the I/O request, message -38 is not delivered. For details about system message -38, see the *Guardian Procedure Errors and Messages Manual*.

## RETURN VALUES

The **SPT_CANCEL( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_close** - Initiates thread-aware **close( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_close(**
**int** *filedes***);**

## PARAMETERS

See the **close(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **close( )** function. Use **spt_close( )** instead of **close( )** to ensure proper operation of the various thread-aware IO functions.

For C applications, a macro to map **close( )** to **spt_close( )** is available when you use the **#define SPT_THREAD_AWARE** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **close( )** to **spt_close( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

## RETURN VALUES

See the **close(2)** reference page.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_closex -** Closes a file descriptor (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <unistd.h>**]
**#include <spthread.h>**

**int spt_closex(**
    **int** *filedes*
    **);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **spt_acceptx( )**, **creat( )**, **dup( )**, **spt_dup2x( )**, **spt_fcntlx( )**, **open( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

## DESCRIPTION

The **spt_closex( )** function is a thread-aware version of the **close( )** function. Use **spt_closex( )** instead of **close( )** to ensure proper operation of the various thread-aware input/output functions.

The **spt_closex( )** function closes the file descriptor specified by the *filedes* parameter.

All regions of the file associated with the *filedes* parameter that this process has previously locked with the **spt_fcntlx( )** function are unlocked. This behavior occurs even if the process still has the file open by another file descriptor.

When the last file descriptor associated with an open file descriptor is closed:

- The open file descriptor is freed.
- The last modification time for the file is updated.
- All locks created by **spt_fcntlx( )** for the file are released.
- If the link count of the file is 0 (zero), the space occupied by the file is freed, and the file is no longer accessible.
- If the file is a socket, the socket is destroyed.
- If the file is a pipe or FIFO, any data remaining in the pipe or FIFO is discarded.

## NOTES

For C applications, a macro to map **close( )** to **spt_closex( )** is available when you use the **#define SPT_THREAD_AWARE_NONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **close( )** to **spt_closex( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occur, the **spt_closex( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not a valid open file descriptor.

[EIO] An input or output error occurred. The device that the file is stored on might be in the down state, or both processors that provide access to the device might have failed.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Functions: **close(2)**, **exec(2)**, **fcntl(2)**, **getsockopt(2)**, **open(2)**, **pipe(2)**, **setsockopt(2)**, **socket(2)**, **spt_close(2)**, **spt_fcntlx(2)**, **tdm_execve(2)**, **tdm_execvep(2)**.

Files: **signal(4)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with this exception:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_closez -** Initiates close( ) function for thread-aware functions

## LIBRARY

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_closez(**
  **int** *filedes***);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **acceptx( )**, **creat( )**, **dup( )**, **dup2( )**, **spt_dup2x( )**, **spt_fcntlz( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

## DESCRIPTION

Use **spt_closez( )** instead of **close( )** or **spt_closex( )** to ensure proper operation of the various thread-aware I/O functions if **spt_*z( )** function calls are used.

The **spt_closez( )** function closes the file descriptor specified by the *filedes* parameter.

All regions of the file specified by the *filedes* parameter that this process has locked with the **spt_fcntlz( )** function are unlocked by the **spt_closez( )** function. This behavior occurs even if the process still has the file open using a different file descriptior.

When the last file descriptor associated with an open file descriptor is closed:

- The open file descriptor is freed.
- The last modification time for the file is updated.
- If the link count of the file is 0 (zero), the space occupied by the file is freed, and the file is no longer accessible.
- If the file is a socket, the socket is destroyed.
- If the file is a pipe or FIFO, any data remaining in the pipe or FIFO is discarded.

## NOTES

For file descriptors for non-regular files, the **spt_closez( )** function behaves exactly the same as **spt_closex( )**. For file descriptors for regular files, **spt_closez( )** first flushes dirty cache blocks by calling **spt_fsyncz( )**, which is a thread aware function that blocks only the calling thread during its operation. If a thread calls **spt_closez( )** to close a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

This function serializes file operations on an open file. If a thread calls **spt_closez( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **close( )** to **spt_closez( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **close( )** to **spt_closez( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the

following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occur, the **spt_closez( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not a valid open file descriptor.

[EIO] An input or output error occurred. The device that the file is stored on might be in the down state, or both processors that provide access to the device might have failed.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

**RELATED INFORMATION**

See the **close(2)** reference page.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with this exception:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_connect -** Initiates thread-aware **connect( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <spthread.h>**

**int spt_connect(**
**int** *socket***,**
**const struct sockaddr ****address***,**
**size_t** *address_len***);**

## PARAMETERS

See the **connect(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **connect( )** function. The socket must be nonblocking for this function to be thread-aware.

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

The following macro maps **spt_connect( )** to **connect( )** and has been defined in **spthread.h**:

```
#define connect(socket, address, address_len)
        spt_connect(socket, address, address_len)
```

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE
```

## RETURN VALUES

See the **connect(2)** reference page. The following also applies:

- Value **errno** is never set to [EINPROGRESS] or [EALREADY].
- If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** of [EBADF].
- If a signal is received via **pthread_kill( )** and is not blocked, ignored, or handled, -1 is returned with an **errno** of [EINTR].

## ERRORS

See the **connect(2)** reference page.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_connectx -** Connects a socket (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <sys/socket.h>**]
**#include <spthread.h>**

**int spt_connectx(**
    **int** *socket***,**
    **const struct sockaddr *** *address***,**
    **size_t** *address_len*
    **);**

## PARAMETERS

*socket* Specifies the file descriptor for the socket.

*address* Points to a **sockaddr** structure that contains the address of the peer socket. The length and format of the address depend on the address family of the socket.

For **AF_INET** sockets, a pointer to the address structure **sockaddr_in** must be cast as a **struct sockaddr**. For **AF_INET6** sockets, a pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**. For **AF_UNIX** sockets, a pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*address_len* Specifies the length of the **sockaddr** structure pointed to by the *address* parameter.

## DESCRIPTION

The **spt_connectx( )** function is a thread-aware version of the **connect( )** function.

The **spt_connectx( )** function requests that a connection be made on a socket. The **spt_connectx( )** function performs a different action for each of the following types of initiating sockets:

- If the initiating socket is not connection-oriented (has the type **SOCK_DGRAM**), the **spt_connectx( )** function sets the peer address but no connection is made. The peer address identifies the socket where all datagrams are sent by subsequent calls to the **spt_sendx( )** function, and limits the remote sender for subsequent **spt_recvx( )** function calls. Datagram sockets can use the **spt_connectx( )** function multiple times to communicate with different peers.

  If the socket is a datagram socket and *address* is a null address for the protocol, the address for the peer socket is reset.

- If the initiating socket is connection-oriented (has the type **SOCK_STREAM**), the **spt_connectx( )** function attempts to make a connection to the socket specified by the *address* parameter. Sockets of type **SOCK_STREAM** can successfully connect only once.

When a connection cannot be created immediately and **O_NONBLOCK** is not set for the file descriptor of the socket, the **spt_connectx( )** call blocks until one of the following conditions occurs:

- A connection is established.
- A timeout occurs.
- A signal is caught.

If a timeout occurs, the **spt_connectx( )** call fails and **errno** is set to [ETIMEDOUT]; the connection is aborted.

If an **spt_connectx( )** call is interrupted by a signal that is caught while the call is blocked waiting to establish a connection, the **spt_connectx( )** call fails and sets **errno** to [EINTR]; the connection is not aborted and is later established asynchronously.

When a connection cannot be created immediately and **O_NONBLOCK** is set for the file descriptor of the socket, the **spt_connectx( )** call fails and sets **errno** to [EINPROGRESS]; the connection is not aborted and is later established asynchronously. Subsequent calls to the **spt_connectx( )** function for the same socket before the connection is completed will fail and set **errno** to [EALREADY].

**NOTES**

The macro to map **connect( )** to **spt_connectx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **connect( )** to **spt_connectx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

When an asynchronous connection is complete, a call to the **select( )** function indicates that the file descriptor for the socket is ready for writing.

**RETURN VALUES**

Upon successful completion, the **spt_connectx( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

**ERRORS**

If any of these conditions occur, the **spt_connectx( )** function sets **errno** to the corresponding value:

[EACCES] The socket is in the **AF_UNIX** domain and either search permission is denied for a component of the pathname in the **sockaddr** structure, or write access to the specified socket is denied.

[EADDRINUSE]
An attempt was made to establish a connection using addresses that are already in use.

[EADDRNOTAVAIL]
The specified address is not available from this node.

[EAFNOSUPPORT]
Addresses in the specified address family cannot be used with this socket.

[EALREADY] A connection request is already in progress for the specified socket.

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNREFUSED]
One of these conditions occured:

- The specified address is not listening for connections or rejected the attempt to connect.
- The socket bound to the **AF_UNIX** *address* is not using the same transport provider as the *socket*. This condition can occur if the system is running **AF_UNIX** Release 2 software and the socket bound to *address* is not of the same mode as *socket*.
- For **AF_UNIX** Release 1 socket or an **AF_UNIX** Release 2 socket in compatibility mode:
  - The caller attempted to connect a socket that previously had been called by the **listen( )** function with a *backlog* parameter less than or equal to 0 (zero), and
  - There is no pending **accept( )** call to that socket.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EHOSTUNREACH]
The destination host cannot be reached.

[EINPROGRESS]
The socket is marked nonblocking (**O_NONBLOCK** is set) and the requested connection is not yet completed. The connection will be completed asynchronously.

[EINTR] The attempt to connect was interrupted by delivery of a signal. The connection will be completed asynchronously.

[EINVAL] One of the following conditions exists:

- The size specified for the *address_len* parameter is not valid for an address in the address family that is used by this connection.
- The **sockaddr** structure contains an invalid address family.

[EIO] The socket is in the **AF_UNIX** domain and an I/O error occurred during a read or write to the file system.

[EISCONN] The specified socket is connection-oriented and is already connected.

[ELOOP] The socket is in the **AF_UNIX** domain and too many symbolic links were encountered in translating the pathname in the **sockaddr** structure.

[ENAMETOOLONG]
The socket is in the **AF_UNIX** domain and one of the following conditions exists:

- The pathname in the **sockaddr** structure exceeds **PATH_MAX** characters.
- A component of the pathname in the **sockaddr** structure exceeds **NAME_MAX** characters.
- The intermediate result of pathname resolution when a symbolic link is part of the pathname in the **sockaddr** structure exceeds **PATH_MAX** characters.

The **pathconf( )** function can be called to obtain the applicable limits.

[ENETDOWN]
The local interface used to reach the destination is down.

[ENETUNREACH]
No route to the network or host is present.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOENT] The socket is in the **AF_UNIX** domain and one of the following conditions exists:

- A component of the pathname specified in the **sockaddr** structure does not name an existing file.
- The **sockaddr** structure specifies an empty string as a pathname.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOTDIR] The socket is in the **AF_UNIX** domain and a component of the pathname specified in the **sockaddr** structure is not a directory.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EPROTOTYPE]
The specified address has a different type than that of the socket bound to the specified peer address.

[ETIMEDOUT]
The attempt to connect timed out during connection establishment.

**RELATED INFORMATION**

Functions: **accept(2)**, **bind(2)**, **connect(2)**, **getsockname(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **socket(2)**, **spt_connect(2)**, **spt_sendx(2)**, **spt_sendmsgx(2)**, **spt_sendtox(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

The XPG4 specification allows certain behaviors of **connect( )** to be implementer-defined. For an indication of the HP implementation behaviors, see the **connect(2)** reference page either online or in the *Open System Services System Calls Reference Manual*.

## NAME

**SPT_CONTROL -** Performs device-dependent input/output operations

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_CONTROL (**
    **short** *filenum***,**
    **short** *operation***,**
    [ **short** *param* ]**,**
    [ **long** *tag* ]
    **);**

## PARAMETERS

*filenum* specifies the Guardian file number of a Guardian file open instance, identifying the file on which the underlying CONTROL procedure performs an input or output operation.

*operation* specifies a value from 1 through 27 that defines a type of operation to be performed. For tables that list *operation* numbers and the possible *param* values for each, see the description of the CONTROL procedure in the *Guardian Procedure Calls Reference Manual*.

*param* specifies a value that defines the operation to be performed. For tables that list *operation* numbers and the possible *param* values for each, see the description of the CONTROL procedure in the *Guardian Procedure Calls Reference Manual*.

*tag* is for nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **SPT_CONTROL( )** function is the thread-aware version of the Guardian CONTROL procedure. CONTROL is used to perform device-dependent input or output operations.

If the **SPT_CONTROL( )** function is used on a file that is opened for nowait I/O, the function must be completed with a call to the AWAITIO procedure.

The following considerations apply to use on disk files:

Writing EOF to an unstructured file
Writing EOF to an unstructured disk file sets the EOF pointer to the relative byte address indicated by the setting of the next-record pointer and writes the new EOF setting in the file label on disk. (File pointer action for CONTROL operation 2, write EOF.)

File is locked If a CONTROL operation is attempted for a file locked through a *filenum* other than that specified in the call to **SPT_CONTROL( )**, the call is rejected with a "file is locked" error 73. If any record is locked in a file, a call to **SPT_CONTROL( )** to write EOF (operation 2) to that same file will be rejected with a "file is locked" error 73.

The following considerations apply to use on magnetic tapes:

When device is not ready
: If a magnetic tape rewind is performed concurrently with application program execution (that is, a rewind operation other than 6), any attempt to perform a read, write, or control operation to the rewinding tape unit while rewind is taking place results in an error indication. A subsequent call to the FILE_GETINFO_ or FILEINFO procedure shows that an error 100 occurred.

Wait for rewind to complete
: If a magnetic tape rewind operation of 6 (wait for completion) is performed as a nowait operation, the application waits at the call to the AWAITIO procedure for the rewind to complete.

The following considerations apply to use for interprocess communication:

Nonstandard *operation* and *param* values
: You can specify any value for the *operation* and *param* parameters. An application-defined protocol should be established for interpreting nonstandard parameter values.

Process not accepting system messages
: If the object of the control operation is not accepting process CONTROL messages, the call to **SPT_CONTROL( )** completes but a subsequent call to the FILE_GETINFO_ or FILEINFO procedure shows that an error 7 occurred.

Process control
: You can obtain the process identifier of the caller to **SPT_CONTROL( )** in a subsequent call to the FILE_GETRECEIVEINFO_ (or LASTRECEIVE or RECEIVEINFO) procedure.

## RETURN VALUES

The **SPT_CONTROL( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

When device handlers do not allow the operation, Guardian file-system error 2 is returned. For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_dup2x -** Duplicates and controls an open file descriptor (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

```
[#include <unistd.h>]
#include <spthread.h>

int spt_dup2x(
        int filedes,
        int new);
```

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **spt_acceptx( )**, **creat( )**, **dup( )**, **spt_dup2x( )**, **spt_fcntlx( )**, **open( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*new* Specifies the open file descriptor that is returned by the **spt_dup2x( )** function. If this descriptor is already in use, it is first deallocated as if it had been closed.

## DESCRIPTION

The **spt_dup2x( )** function is a thread-aware version of the **dup2( )** function.

The **spt_dup2x( )** function returns a new file descriptor on the open file specified by the *filedes* parameter. If *new* is less than 0 (zero) or greater than or equal to the maximum number of opens permitted, **spt_dup2x( )** returns -1 with **errno** set to [EBADF].

The new file descriptor:

- Is the value specified as the *new* parameter:
  - If *filedes* is a valid file descriptor and is equal to *new*, **spt_dup2x( )** returns *new* without closing it.
  - If *filedes* is not a valid file descriptor, **spt_dup2x( )** returns -1 and does not close *new*.
  - The value returned is equal to the value of *new* upon successful completion, or it is -1 upon failure.
- References the same open file descriptor
- Returns the same file pointer as the original file (that is, both file descriptors share one file pointer if the object is a file)
- Returns the same access mode (read, write, or read/write)
- Returns the same file status flags (that is, both file descriptors share the same file status flags)
- Clears the close-on-exec flag (**FD_CLOEXEC** bit) associated with the new file descriptor so that the file remains open across calls to any function in the **exec**, **tdm_exec**, and **tdm_spawn** sets of functions

## NOTES

The macro to map **dup2( )** to **spt_dup2x( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **dup2( )** to **spt_dup2x( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

The **spt_dup2x( )** function provides an alternative interface to the service provided by the **spt_fcntlx( )** function by using the **F_DUPFD** value of the *request* parameter. The call:

```
fid = spt_dup2x( file1, file2 );
```

is equivalent to:

```
close( file2 );
fid = spt_fcntlx( file1, F_DUPFD, file2 );
```

## RETURN VALUES

Upon successful completion, the **spt_dup2x( )** function returns a new file descriptor. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occur, the **spt_dup2x( )** function sets **errno** to the corresponding value:

[EBADF] One of these conditions exists:

- The *filedes* parameter is not a valid open file descriptor.
- The *new* parameter file descriptor is negative or greater than the maximum number of open file descriptors permitted.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation on an input/output process (such as a terminal server process) that has failed or is in the down state.

- The processor for the disk process of the specified file failed during an input or output operation, and the backup process took over.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

**RELATED INFORMATION**

Functions: **close(2)**, **dup(2)**, **dup2(2)**, **exec(2)**, **fcntl(2)**, **open(2)**, **read(2)**, **spt_fcntlx(2)**, **spt_readx(2)**, **spt_writex(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **write(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fclose -** Initiates thread-aware **fclose( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_fclose(FILE *stream);**

## PARAMETERS

See the **fclose(3)** reference page online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **fclose( )** function. Note that the file descriptor underlying the stream must be nonblocking for this function to be thread-aware.

The following macro maps **spt_fclose( )** to **fclose( )** and has been defined in **spthread.h**:

**#define fclose(stream) spt_fclose(stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **fclose(3)** reference page. The following also applies:

- Value **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying the stream becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fclosex -** Closes a stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

**int spt_fclosex (**
    **FILE ****stream*
    **);**

## PARAMETERS

*stream* Specifies the output or update stream.

## DESCRIPTION

The **spt_fclosex( )** function is the thread-aware version of the **fclose( )** function.

The **spt_fclosex( )** function writes buffered data to the stream specified by the *stream* parameter and then closes the associated file. It is automatically called for all open files when the **exit( )** function is invoked. Any unwritten buffered data for the stream is delivered to the host environment to be written to the file; any unread buffered data is discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated. Any further use of the stream specified by the *stream* parameter causes undefined behavior.

The **spt_fclosex( )** function performs the **close( )** function on the file descriptor associated with the *stream* parameter. If the stream was writable and buffered data was not yet written to the file, it marks the **st_ctime** and **st_mtime** fields of the underlying file for update. If the file is not already at end-of-file (EOF), and is capable of seeking, the file pointer of the underlying open file descriptor is adjusted so that the next operation on the open file descriptor deals with the byte after the last one read from or written to the stream being closed.

## NOTES

The macro to map **fclose( )** to **spt_fclosex( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **fclose( )** to **spt_fclosex( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

Upon successful completion, the **spt_fclosex( )** function returns a value of 0 (zero). Otherwise, EOF is returned, and **errno** is set to indicate the error.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), EOF is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_fclosex( )** function sets **errno** to the value that corresponds to the condition:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying the *stream* parameter and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying the *stream* parameter is not valid.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.

[EINTR] The **spt_fclosex( )** function was interrupted by a signal that was caught.

[EIO] The **TOSTOP tty** local mode causes a background process to get a **SIGTTOU** signal if it attempts to write to the controlling terminal. The **SIGTTOU** signal, if it is not caught or ignored, will cause the process to block in a stopped state. A process in an orphaned process group is not allowed to become stopped, because there is no unprivileged process to unblock it. This condition only applies to operations on **stdio** streams associated with **tty**s.

[EIO] is also associated with driver errors.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

## RELATED INFORMATION

Functions: **close(2)**, **exit(2)**, **fclose(3)**, **fflush(3)**, **fopen(3)**, **setbuf(3)**, **spt_closex(2)**, **spt_fclose(2)**, **spt_fflushx(2)**, **spt_fopenx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fcntlx -** Controls open file descriptors (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
[#include <sys/types.h>]
[#include <unistd.h>]
[#include <fcntl.h>]
#include <spthread.h>

int spt_fcntlx (
        int filedes,
        int request
        [ , int argument1 | , struct flock *argument2 ]
        );
```

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **spt_acceptx( )**, **creat( )**, **dup( )**, **spt_dup2x( )**, **spt_fcntlx( )**, **open( )**, **pipe( )**, **socket( )**, or **socketpair( )** function

*request* Specifies the operation to be performed

*argument1* Specifies a variable that depends on the value of the *request* parameter

*argument2* Specifies a variable that depends on the value of the *request* parameter

## DESCRIPTION

The **spt_fcntlx( )** function is the thread-aware version of the **fcntl( )** function.

The **spt_fcntlx( )** function performs controlling operations on the open file specified by the *filedes* parameter.

Values for the *request* parameter are:

**F_DUPFD** Returns a new file descriptor as listed:

- Returns the lowest-numbered available file descriptor that is greater than or equal to the *argument1* parameter
- References the same open file descriptor
- Returns the same file pointer as the original file (that is, both file descriptors share one file pointer if the object is a file)
- Returns the same access mode (read, write, or read/write)
- Returns the same file status flags (that is, both file descriptors share the same file status flags)
- Clears the close-on-exec flag (**FD_CLOEXEC** bit) associated with the new file descriptor so that the file remains open across calls to any function in the **exec**, **tdm_exec**, or **tdm_spawn** sets of functions

The value **F_DUPFD** is invalid for an OSSTTY or Telserv terminal device. If this value is used in a call that specifies such a device for the *filedes* parameter, the call fails and **errno** is set to [EINVAL].

**F_GETFD** Gets the value of the file descriptor flags, defined in the **fcntl.h** header file, that are associated with the value of the *filedes* parameter. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file. The *argument1* parameter or *argument2* parameter is ignored.

The value **F_GETFD** is invalid for an OSSTTY or Telserv terminal device. If this value is used in a call that specifies such a device for the *filedes* parameter, the call fails and **errno** is set to [EINVAL].

**F_SETFD** Sets the value of the file descriptor flags, defined in the **fcntl.h** header file, that are associated with the *filedes* parameter to the value of the *argument1* parameter.

If the **FD_CLOEXEC** flag in the *argument1* parameter is 0 (zero), the file remains open across calls to any function in the **exec**, **tdm_exec**, and **tdm_spawn** sets of functions; otherwise, the file is closed on successful execution of the next function in an **exec**, **tdm_exec**, or **tdm_spawn** function set. When the **FD_CLOEXEC** flag is set, no other flag can be set in the call.

The value **F_SETFD** is invalid for an OSSTTY or Telserv terminal device. If this value is used in a call that specifies such a device for the *filedes* parameter, the call fails and **errno** is set to [EINVAL].

**F_GETFL** Gets the file status flags and file access modes, defined in the **fcntl.h** header file, for the file referred to by the *filedes* parameter.

You can use the mask **O_ACCMODE** on the return value to extract the file access modes. File status flags and file access modes are associated with the file descriptor and do not affect other file descriptors that refer to the same file with different open file descriptors.

The *argument1* or *argument2* parameter is ignored.

The **O_APPEND**, **O_NONBLOCK**, and **O_SYNC** flags are not returned as set if they were ignored in a previous call using **F_SETFL**.

**F_SETFL** Sets the file status flags **O_APPEND**, **O_NONBLOCK**, and **O_SYNC** for the file to which the *filedes* parameter refers, from the corresponding bits in the *argument1* parameter. Some flags are ignored, depending on the file type:

Table 7−1. Ignored File Status Flags (spt_fcntlx Function)

| File type | Ignored file status flags |
|---|---|
| Directory | **O_APPEND**, **O_NONBLOCK**, **O_SYNC** |
| FIFO, pipe | **O_APPEND**, **O_SYNC** |
| Character special file | **O_APPEND**, **O_SYNC** |
| Regular file | **O_NONBLOCK** |
| Socket | None; however, see **O_ASYNC** note in text. |

These file status flags are always accepted and ignored:

**O_ACCMODE**
**O_CREAT**
**O_EXCL**
**O_TRUNC**

The **O_ASYNC** flag is not supported for sockets. If the **O_ASYNC** flag is used with **F_SETFL**, the **fcntl( )** call fails, and **errno** is set to [EINVAL].

The file access mode is not changed when **F_SETFL** is used.

**F_GETOWN** Gets the process ID or process group ID currently receiving the **SIGURG** signal for a socket. A process group ID is returned as a negative value. A positive value indicates the process ID.

The value **F_GETOWN** is invalid for these calls:

- Guardian use of OSS sockets is not supported. If this value is used in a call from the Guardian environment, the call fails, and **errno** is set to [ENOTOSS].
- If this value is used in a call that specifies anything other than a socket for the *filedes* parameter, the call fails, and **errno** is set to [EINVAL].

**F_SETOWN** Sets the process ID or process group ID to receive the **SIGURG** signal for a socket. A process group ID is specified by supplying it as a negative value in the *argument1* parameter; otherwise, the *argument1* parameter is interpreted as a process ID.

The value **F_SETOWN** is invalid for these calls:

- Guardian use of OSS sockets is not supported. If this value is used in a call from the Guardian environment, the call fails, and **errno** is set to [ENOTOSS].
- If this value is used in a call that specifies anything other than a socket for the *filedes* parameter, the call fails, and **errno** is set to [EINVAL].

These values listed for the *request* parameter are available for advisory record locking on regular files. Advisory record locking is supported only for regular files. If attempted on other files, the operation fails, and **errno** is set to [EINVAL].

**F_GETLK** Gets the first lock that blocks the lock description pointed to by the *argument2* parameter. The information retrieved overwrites the information passed to the **fcntl( )** function in the **flock** structure. If no lock is found that would prevent this lock from being created, the structure is left unchanged except for the lock type, which is set to **F_UNLCK**.

**F_SETLK** Sets or clears a file segment lock according to the lock description pointed to by the *argument2* parameter. **F_SETLK** is used to establish shared locks (**F_RDLCK**) or exclusive locks (**F_WRLCK**) and, additionally, to remove either type of lock (**F_UNLCK**). If a shared (read) or exclusive (write) lock cannot be set, the **fcntl( )** function returns immediately with the value -1.

**F_SETLKW** Same as **F_SETLK** except that, if a shared or exclusive lock is blocked by other locks, the process waits until it is unblocked. If a signal is received while **fcntl( )** is waiting for a region, the function is interrupted, -1 is returned, and **errno** is set to [EINTR].

The **O_NONBLOCK** file status flag affects only operations against file descriptors derived from the same **open( )** function.

When a shared lock is set on a segment of a file, other processes can set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock fails if the file descriptor is not opened with read access.

An exclusive lock prevents any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock fails if the file descriptor was not opened with write access.

The **flock** structure describes the type (**l_type** field), starting offset (**l_whence**), relative offset (**l_start**), size (**l_len**), and process ID (**l_pid**) of the segment of the file to be affected.

The value of **l_whence** is set to **SEEK_SET**, **SEEK_CUR**, or **SEEK_END** to indicate that the relative offset of **l_start** bytes is measured from the start of the file, from the current position, or from the end of the file, respectively. The value of **l_len** is the number of consecutive bytes to be locked. The **l_len** value can be negative (where the definition of type **off_t** permits negative values of **l_len**). The **l_pid** field is used only with **F_GETLK** to return the process ID of the process holding a blocking lock. After a successful **F_GETLK** request, the value of **l_whence** becomes **SEEK_SET**.

If **l_len** is positive, the area affected starts at **l_start** and ends at **l_start** + **l_len - 1**. If **l_len** is negative, the area affected starts at **l_start** + **l_len** and ends at **l_start - 1**. Lock lengths can be negative.

Locks can start and extend beyond the current end of a file, but they cannot be negative relative to the beginning of the file. If **l_len** is set to 0 (zero), a lock can be set to always extend to the largest possible value of the file offset for that file. If such a lock also has **l_start** set to 0 (zero) and **l_whence** is set to **SEEK_SET**, the whole file is locked.

Changing or unlocking a portion from the middle of a larger locked segment leaves a smaller segment at either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or when the process holding that file descriptor terminates. Locks are not inherited by a child process in a **fork( )**, **tdm_fork( )**, or **tdm_spawn( )**-type function.

## RETURN VALUES

Upon successful completion, the value returned by the **spt_fcntlx( )** function depends on the value of the *request* parameter, listed:

| | |
|---|---|
| **F_DUPFD** | Returns a new file descriptor. |
| **F_GETFD** | Returns the value of the file descriptor flags. The return value is not negative. |
| **F_GETFL** | Returns the value of file status flags and access modes. The return value is not negative. |
| **F_GETLK** | Returns the value 0 (zero). |
| **F_GETOWN** | Returns the process ID or process group ID of the socket receiving a **SIGURG** signal. A positive value is a process ID; a negative value is a process group ID. |
| **F_SETFD** | Returns the value 0 (zero). |
| **F_SETFL** | Returns the value 0 (zero). |
| **F_SETLK** | Returns the value 0 (zero). |
| **F_SETLKW** | Returns the value 0 (zero). |

**F_SETOWN** Returns the value 0 (zero).

If the **spt_fcntlx( )** function fails, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occur, the **spt_fcntlx( )** function sets **errno** to the corresponding value:

[EAGAIN] The *request* parameter is **F_SETLK**, the type of lock (**l_type**) is shared (**F_RDLCK**) or exclusive (**F_WRLCK**), and a segment of a file to be locked is already exclusive-locked by another process.

The *request* parameter is **F_SETLK**, the type of lock is exclusive, and some portion of a segment of a file to be locked is already shared-locked or exclusive-locked by another process.

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function (such as **spt_writez( )**) is in progress on a regular file and a function that is process-blocking for regular files (such as **read( )**, **spt_read( )**, or **spt_readx( )**) attempts to begin an I/O operation on the same open file.

[EBADF] One of these conditions exists:

- The *request* parameter is **F_SETLK** or **F_SETLKW**, the type of lock is shared (**F_RDLCK**), and *filedes* is not a valid file descriptor open for reading.
- The type of lock is exclusive (**F_WRLCK**), and *filedes* is not a valid file descriptor open for writing.
- The *filedes* parameter is not a valid open file descriptor.

[ECONNRESET]

One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The *argument2* parameter is an invalid address.

[EINTR] The *request* parameter is **F_SETLKW**, and the **spt_fcntlx( )** function was interrupted by a signal that was caught.

[EINVAL] One of these conditions exists:

- The *request* parameter is **F_DUPFD**, and the *argument1* parameter is negative or greater than or equal to the maximum number of opens permitted.
- The *request* parameter is **F_GETLK**, **F_SETLK**, or **F_SETLKW**, and the data pointed to by *argument2* is invalid, or *filedes* refers to a file that does not support locking.

- The *request* parameter is **F_GETOWN**, and the *filedes* parameter does not specify a socket.
- The *request* parameter is **F_SETFD**, and a flag in addition to **FD_CLOEXEC** in the *argument1* parameter is set. When the *request* parameter is **F_SETFD** and **FD_CLOEXEC** is set, no other flag can be set.
- The *request* parameter is **F_SETFL**, and any file status flag other than **O_NONBLOCK**, **O_APPEND**, **O_CREAT**, **O_EXCL**, **O_SYNC**, or **O_TRUNC** is set. (Values set in the **O_ACCMODE** mask are ignored.)
- The *request* parameter is **F_SETOWN**, and the *filedes* parameter does not specify a socket.
- The call attempted to set an advisory record lock on a file that is not a regular file.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[EMFILE] The *request* parameter is **F_DUPFD** and the maximum number of open file descriptors permitted are currently open in the calling process, or no file descriptors greater than or equal to *argument1* are available.

[ENETDOWN]
The *request* parameter is **F_SETLK**, the *filedes* parameter specifies a file on a remote node, and communication with the remote node has been lost.

[ENOLCK] The *request* parameter is **F_SETLK** or **F_SETLKW**, and satisfying the lock or unlock request would cause the number of locked regions in the system to exceed a system-imposed limit.

[ENOTOSS] The *filedes* parameter specifies a socket, and the calling process is running in the Guardian environment. You cannnot use **spt_fcntlx( )** function on an OSS socket from the Guardian environment.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and the backup process took over.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure*

*Errors and Messages Manual*.

## RELATED INFORMATION

Functions: **creat(2)**, **close(2)**, **dup(2)**, **dup2(2)**, **exec(2)**, **fcntl(2)**, **open(2)**, **read(2)**, **socket(2)**, **spt_dup2x(2)**, **spt_readx(2)**, **spt_writex(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **write(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with this exception:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.

The POSIX standard allows certain behaviors of **fcntl( )** to be implementer-defined. For an indication of the HP implementation behaviors, see the **fcntl(2)** reference page either online or in the *Open System Services System Calls Reference Manual*.

## NAME

**spt_fcntlz -** Controls open file descriptors (thread-aware version)

## LIBRARY

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_fcntlz(**
**int** *filedes***,**
**int** *request*
[ **, int** *argument1* |
[**, struct flock** ***argument2* |
**, struct flock64** ***argument2* ]] **);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **spt_acceptx( )**, **creat( )**, **creat64( )**, **dup( )**, **spt_dup2x( )**, **spt_fcntlz( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*request* Specifies the operation to be performed.

*argument1* Specifies a variable that depends on the value of the *request* parameter.

*argument2* Specifies a variable that depends on the value of the *request* parameter.

## DESCRIPTION

The **spt_fcntlz( )** function is a thread aware version of the **fcntl( )** function for file descriptors for non-regular files and for regular files.

The **spt_fcntlz( )** function performs controlling operations on the open file specified by the *filedes* parameter.

Values for the *request* parameter are:

**F_DUPFD** Returns a new file descriptor as listed:

- Returns the lowest-numbered available file descriptor that is greater than or equal to the *argument1* parameter.
- References the same open file description as the original file descriptor.
- Returns the same file pointer as the original file (that is, both file descriptors share one file pointer if the object is a file).
- Returns the same access mode (read, write, or read/write).
- Returns the same file status flags (that is, both file descriptors share the same file status flags).
- Clears the close-on-exec flag (**FD_CLOEXEC** bit) associated with the new file descriptor so that the file remains open across calls to any function in the **exec**, **tdm_exec**, or **tdm_spawn** sets of functions.

The value **F_DUPFD** is invalid for an OSSTTY or Telserv terminal device. If this value is used in a call that specifies such a device for the *filedes* parameter, the call fails and **errno** is set to [EINVAL].

**F_GETFD** Gets the value of the file descriptor flags, defined in the **fcntl.h** header file, that are associated with the value of the *filedes* parameter. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file. The *argument1* parameter or *argument2* parameter is ignored.

The value **F_GETFD** is invalid for an OSSTTY or Telserv terminal device. If this value is used in a call that specifies such a device for the *filedes* parameter, the call fails and **errno** is set to [EINVAL].

**F_SETFD** Sets the value of the file descriptor flags, defined in the **fcntl.h** header file, that are associated with the *filedes* parameter to the value of the *argument1* parameter.

If the **FD_CLOEXEC** flag in the *argument1* parameter is 0 (zero), the file remains open across calls to any function in the **exec**, **tdm_exec**, and **tdm_spawn** sets of functions; otherwise, the file is closed on successful execution of the next function in an **exec**, **tdm_exec**, or **tdm_spawn** function set. When the **FD_CLOEXEC** flag is set, no other flag can be set in the call.

The value **F_SETFD** is invalid for an OSSTTY or Telserv terminal device. If this value is used in a call that specifies such a device for the *filedes* parameter, the call fails and **errno** is set to [EINVAL].

**F_GETFL** Gets the file status flags and file access modes, defined in the **fcntl.h** header file, for the file referred to by the *filedes* parameter.

The file access modes can be extracted by using the mask **O_ACCMODE** on the return value. File status flags and file access modes are associated with the file descriptor and do not affect other file descriptors that refer to the same file with different open file descriptors.

The *argument1* or *argument2* parameter is ignored.

The **O_APPEND**, **O_NONBLOCK**, and **O_SYNC** flags are not returned as set if they were ignored in a previous call using **F_SETFL**.

**F_SETFL** Sets the file status flags **O_APPEND**, **O_NONBLOCK**, and **O_SYNC** for the file to which the *filedes* parameter refers, from the corresponding bits in the *argument1* parameter. Some flags are ignored, depending on the file type, as listed:

Table 7−2. Ignored File Status Flags

| File type | Ignored file status flags |
|---|---|
| Directory | **O_APPEND**, **O_NONBLOCK**, **O_SYNC** |
| FIFO, pipe | **O_APPEND**, **O_SYNC** |
| Character special file | **O_APPEND**, **O_SYNC** |
| Regular file | **O_NONBLOCK** |
| Socket | **O_APPEND, O_SYNC** |

These file status flags are always accepted and ignored:

**O_ACCMODE**
**O_CREAT**
**O_EXCL**
**O_TRUNC**

The file access mode is not changed when **F_SETFL** is used.

**F_GETOWN** Gets the process ID or process group ID currently receiving the **SIGURG** signal for a socket. A process group ID is returned as a negative value. A positive value indicates the process ID.

The value **F_GETOWN** is invalid for these calls:

- Guardian use of OSS sockets is not supported. If this value is used in a call from the Guardian environment, the call fails, and **errno** is set to [ENOTOSS].
- If this value is used in a call that specifies anything other than a socket for the *filedes* parameter, the call fails, and **errno** is set to [EINVAL].

**F_SETOWN** Sets the process ID or process group ID to receive the **SIGURG** signal for a socket. A process group ID is specified by supplying it as a negative value in the *argument1* parameter; otherwise, the *argument1* parameter is interpreted as a process ID.

The value **F_SETOWN** is invalid for these calls:

- Guardian use of OSS sockets is not supported. If this value is used in a call from the Guardian environment, the call fails, and **errno** is set to [ENOTOSS].
- If this value is used in a call that specifies anything other than a socket for the *filedes* parameter, the call fails, and **errno** is set to [EINVAL].

These values listed for the *request* parameter are available for advisory record locking on regular files. Advisory record locking is supported only for regular files. If attempted on other files, the operation fails, and **errno** is set to [EINVAL].

**F_GETLK** Gets the first lock that blocks the lock description pointed to by the *argument2* parameter. The information retrieved overwrites the information passed to the **spt_fcntlz( )** function in the **flock** structure. If no lock is found that would prevent this lock from being created, the structure is left unchanged except for the lock type, which is set to **F_UNLCK**.

**F_GETLK64** Similar to **F_GETLK**, except that it takes a pointer to a **flock64** structure instead of a pointer to a **flock** structure.

**F_SETLK** Sets or clears a file segment lock according to the lock description pointed to by the *argument2* parameter. **F_SETLK** is used to establish shared locks (**F_RDLCK**) or exclusive locks (**F_WRLCK**) and, additionally, to remove either type of lock (**F_UNLCK**). If a shared (read) or exclusive (write) lock cannot be set, the **spt_fcntlz( )** function returns immediately with the value -1.

**F_SETLK64** Similar to **F_SETLK**, except that it takes a pointer to a **flock64** structure instead of a pointer to a **flock** structure.

**F_SETLKW** Same as **F_SETLK** except that, if a shared or exclusive lock is blocked by other locks, the thread waits until it is unblocked. If a signal is received while **spt_fcntlz( )** is waiting for a region, the function is interrupted, -1 is returned, and **errno** is set to [EINTR].

**F_SETLKW64**
Similar to **F_SETLKW**, except that it takes a pointer to a **flock64** structure instead of a pointer to a **flock** structure.

The **O_NONBLOCK** file status flag affects only operations against file descriptors derived from the same **open( )** function.

When a shared lock is set on a segment of a file, other processes can set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock fails if the file descriptor is not opened with read access.

An exclusive lock prevents any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock fails if the file descriptor was not opened with write access.

The **flock** and **flock64** structures describe the type (**l_type** field), starting offset (**l_whence**), relative offset (**l_start**), size (**l_len**), and process ID (**l_pid**) of the segment of the file to be affected.

The value of **l_whence** is set to **SEEK_SET**, **SEEK_CUR**, or **SEEK_END** to indicate that the relative offset of **l_start** bytes is measured from the start of the file, from the current position, or from the end of the file, respectively. The value of **l_len** is the number of consecutive bytes to be locked. The **l_len** value can be negative (where the definition of type **off_t** permits negative values of **l_len**). The **l_pid** field is used only with **F_GETLK** or **F_GETLK64** to return the process ID of the process holding a blocking lock. After a successful **F_GETLK** or **F_GETLK64** request, the value of **l_whence** becomes **SEEK_SET**.

If **l_len** is positive, the area affected starts at **l_start** and ends at **l_start** + **l_len - 1**. If **l_len** is negative, the area affected starts at **l_start** + **l_len** and ends at **l_start - 1**. Lock lengths can be negative.

Locks can start and extend beyond the current end of a file, but they cannot be negative relative to the beginning of the file. If **l_len** is set to 0 (zero), a lock can be set to always extend to the largest possible value of the file offset for that file. If such a lock also has **l_start** set to 0 (zero) and **l_whence** is set to **SEEK_SET**, the whole file is locked.

Changing or unlocking a portion from the middle of a larger locked segment leaves a smaller segment at either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or when the process holding that file descriptor terminates. Locks are not inherited by a child process in a function like **fork( )**, **tdm_fork( )**, or **tdm_spawn( )**.

## NOTES

For file descriptors for non-regular files, the **spt_fcntlz( )** function behaves exactly the same as the **spt_fcntlx( )** function. For regular files, if the **spt_fcntlz( )** function needs to wait for F_SETLKW or F_SETLKW64 requests, **spt_fcntlz( )** blocks the thread that called the function (instead of blocking the entire process).

This function serializes file operations on an open file. If a thread calls **spt_fcntlz( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **fcntl( )** to **spt_fcntlz( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **fcntl( )** to **spt_fcntlz( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

Upon successful completion, the value returned by the **spt_fcntlz( )** function depends on the value of the *request* parameter, listed:

**F_DUPFD** Returns a new file descriptor.

**F_GETFD** Returns the value of the file descriptor flags. The return value is not negative.

**F_GETFL** Returns the value of file status flags and access modes. The return value is not negative.

**F_GETLK** Returns the value 0 (zero).

**F_GETLK64** Returns the value 0 (zero).

**F_GETOWN** Returns the process ID or process group ID of the socket receiving a **SIGURG** signal. A positive value is a process ID; a negative value is a process group ID.

**F_SETFD** Returns the value 0 (zero).

**F_SETFL** Returns the value 0 (zero).

**F_SETLK** Returns the value 0 (zero).

**F_SETLK64** Returns the value 0 (zero).

**F_SETLKW** Returns the value 0 (zero).

**F_SETLKW64**
Returns the value 0 (zero).

**F_SETOWN** Returns the value 0 (zero).

If the **spt_fcntlz( )** function fails, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **spt_fcntlz( )** function sets **errno** to the corresponding value:

[EAGAIN] The *request* parameter is **F_SETLK** or **F_SETLK64**, the type of lock (**l_type**) is shared (**F_RDLCK**) or exclusive (**F_WRLCK**), and a segment of a file to be locked is already exclusive-locked by another process.

The *request* parameter is is **F_SETLK** or **F_SETLK64**, the type of lock is exclusive, and some portion of a segment of a file to be locked is already shared-locked or exclusive-locked by another process.

[EBADF] One of these conditions exists:

- The *request* parameter is **F_SETLK**, **F_SETLK64**, **F_SETLKW**,or **F_SETLKW64**, the type of lock is shared (**F_RDLCK**), and *filedes* is not a valid file descriptor open for reading.
- The type of lock is exclusive (**F_WRLCK**), and *filedes* is not a valid file descriptor open for writing.
- The *filedes* parameter is not a valid open file descriptor.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The peer socket forcibly closed the connection.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The *argument2* parameter is an invalid address.

[EINTR] The *request* parameter is **F_SETLKW** or **F_SETLKW64**, and the **spt_fcntlz( )** function was interrupted by a signal that was caught.

[EINVAL] One of these conditions exists:

- The *request* parameter is **F_DUPFD**, and the *argument1* parameter is negative or greater than or equal to the maximum number of open file descriptors permitted.
- The *request* parameter is **F_GETLK**, **F_GETLK64**, **F_SETLK**, **F_SETLK64**, **F_SETLKW**,or **F_SETLKW64**, and the data pointed to by *argument2* is invalid, or *filedes* refers to a file that does not support locking.
- The *request* parameter is **F_GETOWN**, and the *filedes* parameter does not specify a socket.
- The *request* parameter is **F_SETFD**, and a flag in addition to **FD_CLOEXEC** in the *argument1* parameter is set. When the *request* parameter is **F_SETFD** and **FD_CLOEXEC** is set, no other flag can be set.
- The *request* parameter is **F_SETFL**, and any file status flag other than **O_NONBLOCK**, **O_APPEND**, **O_CREAT**, **O_EXCL**, **O_SYNC**, or **O_TRUNC** is set. (Values set in the **O_ACCMODE** mask are ignored.)
- The *request* parameter is **F_SETOWN**, and the *filedes* parameter does not specify a socket.
- The call attempted to set an advisory record lock on a file that is not a regular file.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[EMFILE] The *request* parameter is **F_DUPFD** and the maximum number of open file descriptors permitted are currently open in the calling process, or no file descriptors greater than or equal to *argument1* are available.

[ENETDOWN]
The *request* parameter is **F_SETLK** or **F_SETLK64**, the *filedes* parameter specifies a file on a remote node, and communication with the remote node has been lost.

[ENOLCK] The *request* parameter is **F_SETLK**, **F_SETLK64**, **F_SETLKW**, or **F_SETLKW64**, and satisfying the lock or unlock request would cause the number of locked regions in the system to exceed a system-imposed limit.

[ENOTOSS] One of these conditions occurred:

- The *filedes* parameter specifies a socket, and the calling process is running in the Guardian environment. You cannot use the **spt_fcntlz( )** function on an OSS socket from the Guardian environment.
- The calling process is running in the Guardian environmennt and the *request* parameter is **F_SETLK**, **F_SETLK64**, **F_SETLKW**, or **F_SETLKW64**.

[EOVERFLOW]
The command argument is **F_GETLK**, **F_SETLK**, or **F_SETLKW**, and the smallest offset (if *l_len* parameter is zero), or the highest offset (if the *l_len* parameter is nonzero), of any byte in the requested segment cannot be represented correctly in an object of type **off_t**.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and the backup process took over.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Functions: **creat(2)**, **creat64(2)**, **close(2)**, **dup(2)**, **dup2(2)**, **exec(2)**, **fcntl(2)**, **open(2)**, **open64(2)**, **read(2)**, **socket(2)**, **spt_dup2x(2)**, **spt_readx(2)**, **spt_readz(2)**, **spt_writex(2)**, **spt_writez(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **write(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with this exception:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.
- The **spt_fcntlz( )** function does not return the **errno** value [EDEADLK].
- The **spt_fcntlz( )** function does not support the **O_ASYNC** flag.

The POSIX standards allow some features of the **fcntl( )** function to be defined by the implementer. For more information see the **fcntl(2)** reference page.

## NAME

**spt_fd_read_ready** - Waits on read-ready file descriptor

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

```
#include <spthread.h>

int spt_fd_read_ready(
        const int fd,
        struct timeval *timeout);
```

## PARAMETERS

*fd* Specifies an OSS file descriptor.

*timeout* On input, the maximum interval to wait for *fd* ready; if NULL, then no timeout will occur. On output, the interval remaining.

## DESCRIPTION

Waits on a file descriptor to be read-ready or have an exception pending.

## RETURN VALUES

0 (zero) No error.

[EINTR] A signal was received via **pthread_kill( )** and is not blocked, ignored, or handled.

[EINVAL] Invalid function argument.

[EBADF] File descriptor not open for reading or closed while being waited on.

[ENOTSUP] Operation not supported on file descriptor.

[ETIMEDOUT] The timeout has occurred.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fd_write_ready -** Waits on write-ready file descriptor

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_fd_write_ready(**
**const int** *fd***,**
**struct timeval ****timeout***);**

## PARAMETERS

*fd* Specifies an OSS file descriptor.

*timeout* On input, specifies the maximum interval to wait for *fd* ready.

If NULL, specifies that no *timeout* will occur.

On output, specifies the interval remaining.

## DESCRIPTION

Wait on a file descriptor to be write-ready or have an exception pending.

## RETURN VALUES

0 (zero) No error.

[EINTR] A signal was received via **pthread_kill( )** and is not blocked, ignored, or handled.

[EINVAL] Invalid function argument.

[EBADF] File descriptor was not open for writing or was closed while being waited on.

[ENOTSUP] Operation was not supported on file descriptor.

[ETIMEDOUT]
*timeout* has occurred.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fflush -** Initiates thread-aware **fflush( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_fflush(**
**FILE ****stream***);**

## PARAMETERS

See the **fflush(3)** reference page either online or in the *Guardian C Native Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **fflush(|)** function. The file descriptor underlying the stream must be nonblocking for this function to be thread-aware.

The following macro maps **spt_fflush(2)** to **fflush(3)** and has been defined in **spthread.h**:

**#define fflush(stream) spt_fflush(stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **fflush(3)** reference page. The following also applies:

- Value **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying stream becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fflushx -** Flushes a stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

**int spt_fflushx(**
**FILE ****stream*
**);**

## PARAMETERS

*stream* Specifies the output or update stream.

## DESCRIPTION

The **spt_fflushx( )** function is the thread-aware version of the **fflush( )** function.

The **spt_fflushx( )** function writes any buffered data for the stream specified by the *stream* parameter and leaves the stream open. If *stream* is a null pointer, the **spt_fflushx( )** function performs this flushing action on all streams for which the behavior was previously defined. The **st_ctime** and **st_mtime** fields of the underlying file are marked for update.

## NOTES

The macro to map **fflush( )** to **spt_fflushx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **fflush( )** to **spt_fflushx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

Upon successful completion, the **spt_fflushx( )** function returns a value of 0 (zero). Otherwise, EOF is returned, and **errno** is set to indicate the error.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), EOF is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_fflushx( )** function sets **errno** to the value that corresponds to the condition.

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying the *stream* parameter is not valid.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.

[EINTR] The **spt_fflushx( )** function was interrupted by a signal that was caught.

[EIO] The **TOSTOP tty** local mode causes a background process to get a **SIGTTOU** signal if it attempts to write to the controlling terminal. The **SIGTTOU** signal, if it is not caught or ignored, will cause the process to block in a stopped state. A process in an orphaned process group is not allowed to become stopped, because there is no unprivileged process to unblock it. This condition only applies to operations on **stdio** streams associated with **tty**s.

[EIO] is also associated with driver errors.

[ENOSPC] No free space was remaining on the device containing the file.

[EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

## RELATED INFORMATION

Functions: **close(2)**, **exit(2)**, **fclose(3)**, **fflush(3)**, **fopen(3)**, **setbuf(3)**, **spt_fclosex(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fgetc -** Initiates thread-aware **fgetc( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_fgetc(**
    **FILE ***stream***);**

## PARAMETERS

See the **fgetc(3)** reference page either online or in the *Guardian C Native Library Calls Reference Manual*.

## DESCRIPTION

Thread-aware **fgetc(3)**. The file descriptor underlying the stream must be nonblocking for this function to be thread aware.

The following macro maps **spt_fgetc(|)** to **fgetc(|)** and has been defined in **spthread.h**:

**#define fgetc(stream) spt_fgetc(stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See **fgetc(3)** reference page. The following also applies:

- Value **errno** is never set to EAGAIN or EWOULDBLOCK.
- If the file descriptor underlying stream becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill(|)** function and is not blocked, ignored, or handled, EOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fgetcx -** Gets a character from a specified input stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

```
int spt_fgetcx (
        FILE *stream
        );
```

## PARAMETERS

*stream* Points to the file structure of an open file.

## DESCRIPTION

The **spt_fgetcx( )** function is the thread-aware version of the **fgetc( )** function.

The **spt_fgetcx( )** function returns the next byte from the input specified by the *stream* parameter and moves the file pointer, if defined, ahead one byte in *stream*.

## NOTES

The macro to map **fgetc( )** to **spt_fgetcx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **fgetc( )** to **spt_fgetcx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

The **spt_fgetcx( )** function returns a character if successful. It returns the integer constant **EOF** at the end of the file or upon an error. The function sets **errno** when an error is encountered.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), EOF is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_fgetcx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the underlying input stream and the process would be delayed by the read operation.

[EBADF] The file descriptor underlying the input stream is not a valid file descriptor or is not open for reading.

[EINTR] The read operation was interrupted by a signal that was caught and no data was transferred.

[ENXIO] A request was made on a nonexistent device, or the request was outside the capabilities of the device.

[EIO] The call is attempting to read from the process's controlling terminal and either the process is ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.

[ENOMEM] Insufficient memory is available for the operation.

Any error encountered during the underlying call to the **spt_readx( )** function can cause this function to return the corresponding **errno** value reported by the **spt_readx( )** function. If your application program encounters an **errno** value not listed on this reference page, see the **spt_readx(2)** reference page either online or in the *Open System Services System Calls Reference Manual* the cause of that error.

## RELATED INFORMATION

Functions: **fgetc(3)**, **fgetcx(3)**, **getc(3)**, **getchar(3)**, **gets(3)**, **getwc(3)**, **putc(3)**, **read(2)**, **spt_fgetc(2)**, **spt_getcx(2)**, **spt_getcharx(2)**, **spt_getsx(2)**, **spt_getwcx(2)**, **spt_putcx(2)**, **spt_readx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fgets -** Initiates thread-aware **fgets( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

```
char *spt_fgets(
        char *string,
        int n,
        FILE *stream);
```

## PARAMETERS

See the **fgets(3)** reference page either online or in the *Guardian C Native Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **fgets( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread aware.

The following macro maps **spt_fgets( )** to **fgets( )** and has been defined in **spthread.h**:

**#define fgets(string, n, stream) spt_fgets(string, n, stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **fgets(3)** reference page. The following also applies:

- Value **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying stream becomes invalid (is closed by another thread), NULL is returned with an **errno** of [EBADF].
- If a signal is received via **pthread_kill(2)** and is not blocked, ignored, or handled, NULL is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fgetsx -** Gets a string from a stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

```
[#include <stdio.h>]
#include <spthread.h>

char *spt_fgetsx (
        char *string,
        int n,
        FILE *stream
        );
```

## PARAMETERS

*string* Points to a string to receive bytes.

*n* Specifies an upper bound on the number of bytes to read.

*stream* Points to the **FILE** structure of an open file.

## DESCRIPTION

The **spt_fgetsx( )** function is the thread-aware version of the **fgets( )** function.

The **spt_fgetsx( )** function reads bytes from the data pointed to by the *stream* parameter into the array pointed to by the *string* parameter. Data is read until *n*-1 bytes have been read, until a new-line character is read and transferred to *string*, or until an end-of-file EOF condition is encountered. The string is then terminated with a NULL character.

## NOTES

The macro to map **fgets( )** to **spt_fgetsx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **fgets( )** to **spt_fgetsx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

If the end of the file is encountered and no characters have been read, no characters are transferred to *string* and a null pointer is returned. If a read error occurs, a null pointer is returned. Otherwise, *string* is returned.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), EOF is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_fgetsx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the underlying input stream and the process would be delayed by the read operation.

[EBADF] The file descriptor underlying the input stream is not a valid file descriptor or is not open for reading.

[EINTR] The read operation was interrupted by a signal that was caught and no data was transferred.

[EIO] The call is attempting to read from the process's controlling terminal and either the process is ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.

[ENOMEM] Insufficient memory is available for the operation.

[ENXIO] A request was made on a nonexistent device, or the request was outside the capabilities of the device.

Any error encountered during the underlying call to the **spt_readx( )** function can cause this function to return the corresponding **errno** value reported by the **spt_readx( )** function. If your application program encounters an **errno** value not listed on this reference page, refer to the **spt_readx(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for the cause of that error.

## RELATED INFORMATION

Functions: **clearerr(3)**, **feof(3)**, **ferror(3)**, **fgets(3)**, **fileno(3)**, **fopen(3)**, **fread(3)**, **getc(3)**, **gets(3)**, **getwc(3)**, **puts(3)**, **scanf(3)**, **spt_fgetc(2)**, **spt_getcx(2)**, **spt_getcharx(2)**, **spt_getsx(2)**, **spt_getwcx(2)**, **spt_putsx(2)**, **spt_readx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fgetwc -** Initiates thread-aware **fgetwc( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_fgetwc(**
**FILE ****stream***);**

## PARAMETERS

See the **fgetwc(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **fgetwc( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread aware.

The following macro maps **spt_fgetwc( )** to **fgetwc( )** and has been defined in **spthread.h**:

**#define fgetwc(stream) spt_fgetwc(stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **fgetwc(3)** reference page. The following also applies:

- Value **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying stream becomes invalid (is closed by another thread), WEOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, WEOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fgetwcx -** Gets a wide character from a a specified input stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <wchar.h>**]
**#include <spthread.h>**

**wint_t spt_fgetwcx (**
    **FILE ****stream*
    **);**

## PARAMETERS

*stream* Specifies the input data.

## DESCRIPTION

The **spt_fgetwcx( )** function is the thread-aware version of the **fgetwc( )** function.

The **spt_fgetwcx( )** function gets the next wide character from the input stream specified by the *stream* parameter.

## NOTES

The macro to map **fgetwc( )** to **spt_fgetwcx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **fgetwc( )** to **spt_fgetwcx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

This function returns the wide character read or the constant **WEOF** (wide-character end-of-file) at the end of the file or upon an error.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), **WEOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **WEOF** is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_getwcx( )** function sets **errno** to the corresponding value:

[EBADF] The file descriptor underlying *stream* is no longer valid.

[EINTR] A signal was received that is not blocked, ignored, or handled.

## RELATED INFORMATION

Functions: **fgetwc(3)**, **fopen(3)**, **fread(3)**, **getc(3)**, **gets(3)**, **getwc(3)**, **getwchar(3)**, **putwc(3)**, **scanf(3)**, **spt_fgetwc(2)**, **spt_freadx(2)**, **spt_getcx(2)**, **spt_getsx(2)**, **spt_getwcx(2)**, **spt_getwcharx(2)**, **spt_putwcx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_FILE_CLOSE_ -** Closes an open Guardian file

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_FILE_CLOSE_ (**
    **short** *filenum***,**
    [**short** *tape_disposition* ]
    **);**

## PARAMETERS

*filenum* specifies the file number of a Guardian file open instance that identifies the file to be closed.

*tape_disposition*
Indicates the tape control action to take:

| | |
|---|---|
| 0 | Rewind and unload; do not wait for completion. |
| 1 | Rewind and unload, do not wait for completion. |
| 2 | Rewind and leave online; do not wait for completion. |
| 3 | Rewind and leave online; wait for completion. |
| 4 | Do not rewind; leave online. |

Other input values result in no error if the file is a tape device; the control action might be unpredictable. If this parameter is omitted, 0 (zero) is used.

## DESCRIPTION

The **SPT_FILE_CLOSE_ ()** function is the thread-aware version of the Guardian FILE_CLOSE_ procedure.

The FILE_CLOSE_ procedure closes a Guardian file open instance. Closing a file open instance terminates access to the file through that open instance. You can use **SPT_FILE_CLOSE_()** to close files that were opened by **SPT_FILE_OPEN_()**.

For programming information about the FILE_CLOSE_ procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

### Considerations

Returning space allocation after closing a file
Closing a disk file causes the space that is used by the resident file control block to be returned to the system main-memory pool if the disk file is not open concurrently. A temporary disk file is purged if the file was not open concurrently. Any space that is allocated to that file is made available for other files. With any file closure, the space allocated to the access control block (ACB) is returned to the system.

Closing a nowait file open

If an **SPT_FILE_CLOSE_( )** call is executed for a nowait file that has pending operations, any incomplete operations are canceled. There is no indication as to whether the operation completed or not.

Labeled tape processing

If your system has labeled tape processing enabled, all tape actions (as specified by *tape_disposition*) wait for completion.

Process close message

A process can receive a process close system message when it is closed by another process. It can obtain the process handle of the closer by a subsequent call to the Guardian FILE_GETRECEIVEINFO_ procedure. For detailed information about system messages, see the *Guardian Procedure Errors and Messages Manual*.

This message is also received if the close is made by the backup process of a process pair. Therefore, a process can expect two of these messages when being closed by a process pair.

**RETURN VALUES**

The **SPT_FILE_CLOSE_ ( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

**ERRORS**

None. This function does not set the **errno** variable.

**RELATED INFORMATION**

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEX(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

### NAME

**spt_FileIOHandler_p -** Executes callback type required by **spt_regFileIOHandler( )**

### LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

### SYNOPSIS

**#include <spthread.h>**

**typedef void (*spt_FileIOHandler_p)(const short** *filenum***,**
**const long** *tag*, const long *count_transferred***,**
**const long** *error***, void ***userdata***);**

### PARAMETERS

*filenum* Specifies Guardian file number whose IO has completed

*tag* Specifies tag of completed IO

*count_transferred*
Specifies transfer count of completed IO

*error* Specifies Guardian error number for completed IO

*userdata* Specifies address of user data area; set when the application called the **spt_awaitio( )** function

### DESCRIPTION

Callback type required by the **spt_regFileIOHandler()** function. The callback is executed in the context of the last running thread; it executes on the stack of the last running thread.

### RETURN VALUES

None.

### STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

**NAME**

**SPT_FILE_OPEN_** - Establishes a communication path between an application process and a file

**LIBRARY**

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

**SYNOPSIS**

[**#include <cextdecs.h>**]
**#include <spthread.h>**

```
short SPT_FILE_OPEN_ (
        { const char *filename | const char *pathname },
        short length,
        short *filenum,
        [ short access ],
        [ short exclusion ],
        [ short nowait_depth ],
        [ short sync_or_receive_depth ],
        [ short options ],
        [ short seq_block_buffer_id ],
        [ short seq_block_buffer_len ],
        [ short *primary_processhandle ],
        [ long elections ]
        );
```

**PARAMETERS**

*filename* | *pathname*

*filename* specifies the Guardian filename of a Guardian file to be opened. The value of *filename* must be a valid fully or partially qualified file name or DEFINE name. If the name is partially qualified, it is resolved using the contents of the =_DEFAULTS DEFINE.

*pathname* specifies the OSS filename or pathname of an OSS file to be opened. The value of the *pathname* parameter is terminated by a null character. *options* bit 10 must be set to 1 to open an OSS file.

*filenum*

returns a Guardian file number that is used to identify the Guardian file open instance in subsequent Guardian file-system calls. If the file cannot be opened, a value of -1 is returned.

*filenum* is used as an input parameter only when you are attempting a backup open. In that case, you must supply the *primary_processhandle* parameter or else the input value of *filenum* is ignored. For a backup open, the value specified for *filenum* must be the *filenum* value that was returned when the file was opened by the primary process. If a backup open is successful, the input value of *filenum* is returned unless *options* bit 3 is set, in which case a new file number is assigned for the backup open. If the backup open is unsuccessful, -1 is returned.

*access*

Specifies the desired access mode for the file to be opened. Valid values are:

| | |
|---|---|
| 0 | Read-write |

| | |
|---|---|
| 1 | Read only |
| 2 | Write only |
| 3 | Extend (supported only for tape) |

The default is 0 (zero).

*exclusion* Specifies the desired mode of compatibility with other openers of the file. Valid values are:

| | |
|---|---|
| 0 | Shared |
| 1 | Exclusive |
| 2 | Process exclusive |
| 3 | Protected |

The default is 0 (zero).

*nowait_depth* Specifies the number of nowait I/O operations that can be in progress for the file concurrently with other processing. If this parameter is omitted or 0 (zero), only waited I/O operations are permitted against the file. The maximum value is 1 for disk files and $RECEIVE. The maximum value is 15 for other objects, except for the Transaction Monitoring Facility (TMF) transaction pseudofile (TFILE), which has a maximum of 1000. For details about the TFILE, see the *TMF Application Programmer's Guide*.

*sync_or_receive_depth*
The purpose of this parameter depends on the type of device being opened:

disk file Specifies the number of nonretryable (that is, write) requests whose completion the Guardian file system must remember. You must specify a value of 1 or greater to recover from a path failure occurring during a write operation. This value also implies the number of write operations that the primary process in a process pair can perform to this file without intervening checkpoints to its backup process. For disk files, this parameter is called sync depth. The maximum value is 15.

If omitted, or if 0 (zero) is specified, internal checkpointing does not occur. Disk path failures are not automatically retried by the file system.

$RECEIVE file
Specifies the maximum number of incoming messages read by the **SPT_READUPDATEX( )** function that the application process is allowed to queue before corresponding reply operations must be performed. If omitted or 0 (zero), **SPT_READUPDATEX( )** and reply operations to $RECEIVE are not permitted. For $RECEIVE, this parameter is called receive depth, and the maximum number of queued incoming messages is 4047.

| | |
|---|---|
| process pair | Specifies whether an I/O operation is automatically redirected to the backup process if the primary process or its processor module fails. For processes, this parameter is called sync depth. The process determines the maximum value. The value must be at least 1 for an I/O operation to a remote process pair to recover from a network failure. If this parameter is greater than or equal to 1, the server is expected to save or be able to regenerate that number of replies. If this parameter is 0 (zero), and if an I/O operation cannot be performed to the primary process of a process pair, an error indication is returned to the originator of the message. On a subsequent I/O operation, the file system redirects the request to the backup process. |

For other device types, the meaning of this parameter depends on whether the sync-ID mechanism is supported by the device being opened. If the device does not support the sync-ID mechanism, 0 (zero) is used regardless of what you specify (this is the most common case). If the device supports the sync-ID mechanism, specifying a nonzero value causes the results of that number of operations to be saved; in case of path failures, the operations are retried automatically. The actual value being used can be obtained by a call to the FILE_GETINFOLIST_ procedure.

*options* Specifies optional characteristics as a bit mask. The bits, when set to 1, indicate:

| | |
|---|---|
| 0 | Unstructured access. For disk files, access is to occur as if the file were unstructured, that is, without regard to record structures and partitioning. (For unstructured files, setting this bit to 1 causes secondary partitions to be inaccessible.) This bit must be 0 (zero) for other devices. |
| 1 | Nowait open processing. Specifies that the processing of the open proceeds in a nowait manner. Unless **SPT_FILE_OPEN_( )** returns an error, a nowait open must be completed by a call to the Guardian AWAITIOX procedure. This option cannot be specified for the TMF transaction pseudofile (TFILE). This option does not determine the nowait mode of I/O operations. The *nowait_depth* parameter, which controls the nowait mode of I/O operations, must have a nonzero value when you use this option. |
| 2 | No open time update. For disk files, the "time of last open" file attribute is not updated by this open. This bit must be 0 (zero) for other devices. |
| 3 | Any file number for backup open. When performing a backup open, specifies that the system can use any file number for the backup open. A value of 0 (zero) specifies that the backup open is to have the same file number as the primary open. Guardian file-system error 12 is returned if that file number is already in use. |
| 4 through 9 | Reserved; specify 0 (zero). |

| | |
|---|---|
| 10 | Open an OSS file by its OSS pathname. Specifies that the file to be opened is identified by the *pathname* parameter. |
| 11 | Reserved; specify 0 (zero). |
| 12 | No transactions. For $RECEIVE, messages are not to include transaction identifiers. This bit must be 0 (zero) if bit 15 is 1. |
| 13 | Internationalization locale support. For $RECEIVE, data messages include internationalization locale information. This bit must be 0 (zero) if bit 15 is 1. For information about internationalization, see the *Software Internationalization Guide*. |
| 14 | Old-format system messages. For $RECEIVE, system messages should be delivered in C-series format. If this bit is 0 (zero), D-series format messages are delivered. For other device types, this bit must be 0 (zero). See **Interprocess Communication Considerations** in the **DESCRIPTION** subsection of this reference page. |
| 15 | No file-management system messages. For $RECEIVE, specifies that the caller does not wish to receive process open, process close, CONTROL, SETMODE, SETPARAM, RESETSYNC, and CONTROLBUF messages. If this bit is 0 (zero), messages are delivered as normal; some messages are received only with **SPT_SETMODE(80)**. For other device types, this bit must be 0 (zero). |

When *options* is omitted, 0 (zero) is used for all bits.

*seq_block_buffer_id*

If present and not 0 (zero), identifies the buffer to be used for shared sequential block buffering; all opens made through **SPT_FILE_OPEN_( )** and using this ID share the same buffer. You can supply any integer value for this parameter.

If *seq_block_buffer_id* is omitted or 0 (zero), and sequential block buffering is requested, the buffer is not shared. In this case, the buffer resides in the process's process file segment (PFS) with the size given by *seq_block_buffer_len*.

*seq_block_buffer_len*

Specifies whether sequential block buffering is being requested. If this parameter is supplied with a value greater than 0 (zero), it indicates a request for sequential block buffering and specifies the length in bytes of the sequential block buffer. If this parameter is omitted or 0 (zero), sequential block buffering is not requested. Sequential block buffering is only for disk files.

If this value is less than the data-block length that was given to this file or to any associated alternate-key file, the larger value is used. Supplying a nonzero value for this parameter causes a buffer to be allocated unless an existing buffer is to be shared (see the *seq_block_buffer_id* parameter). If an existing buffer is to be shared, but it is smaller than *seq_block_buffer_len*, sequential block buffering is not provided and a warning value of 5 is returned.

*primary_processhandle*
Indicates that the caller is requesting a backup open and specifies the process handle of the primary process that already has the file open when its backup attempts to open the file. If this parameter is supplied and not null (a null process handle has -1 in each word), *filenum* must contain the *filenum* value that was returned to the primary. If a null process handle is supplied, or the parameter is omitted, a normal open is being requested. Use this option only when the backup process is the caller. It is more common for the primary process to perform this operation by a call to the FILE_OPEN_CHKPT_ procedure.

*elections*
Specifies the following options as a bit mask:

0 through 30 — Reserved; specify 0 (zero).

31 — Use 64-bit primary keys. For disk files only, bit 31 specifies that 64-bit primary-key values are used instead of 32-bit values for unstructured, relative, or entry-sequenced files. Bit 31 is ignored for key-sequenced files and nondisk devices.

You can use the *elections* parameter with both Format 1 and Format 2 Guardian files. If this parameter is omitted, 0 (zero) is used for all bits.

## DESCRIPTION

The **SPT_FILE_OPEN_( )** function is the thread-aware version of the Guardian FILE_OPEN_ procedure.

The **SPT_FILE_OPEN_( )** function establishes a communication path between an application process and a file. When **SPT_FILE_OPEN_( )** successfully completes, it returns a Guardian file number to the caller. The file number identifies this access path to the file in subsequent Guardian file-system calls.

### General Considerations

File numbers
File numbers are unique within a process. The lowest file number is 0 (zero) and is reserved for $RECEIVE; the remaining file numbers start at 1. The lowest available file number is always assigned, except in the case of backup opens. When a file is closed, its file number becomes available for a subsequent file open to use.

Maximum number of open files
The maximum number of files in the system that can be open at any given time depends on the space available for control blocks: access control blocks (ACBs), file control blocks (FCBs), and open control blocks (OCBs). The amount of space available for control blocks is limited primarily by the physical memory size of the system. The maximum amount of space for ACBs is determined by the size of the process file segment (PFS). See the description of the *pfs-size* parameter for the PROCESS_CREATE_ procedure in the *Guardian Procedure Calls Reference Manual*.

Multiple opens by the same process
If a given file is opened more than once by the same process, a unique file number is returned for each open. These file numbers provide logically separate accesses to the same file; each file number has its own ACB, its own file position, and its own last error value. If a nowait I/O operation haS begun and a second nowait operation is started (using a second file number for the same file), the I/O requests:

- Are independent
- Might arrive in either order at the destination
- Might complete in either order

Multiple opens on a given file can create a deadlock. Locks are granted on an open file (that is, file number) basis. Therefore, if a process opens the same file multiple times, a lock of one file number excludes access to the file through other file numbers. The process is suspended forever if the default locking mode is in effect and a deadlock occurs.

Limit on number of concurrent opens

There is a limit on the total number of concurrent opens permitted on a file. This determination includes opens by all processes. The specific limit for a file depends on the file's device type:

| | |
|---|---|
| Disk files | Cannot exceed 32,767 opens per disk. |
| Process | Defined by the process (see the discussion of controlling openers in the *Guardian Programmer's Guide*). |
| $0 | Unlimited opens. |
| $0.#ZSPI | 128 concurrent opens permitted. |
| $OSP | Ten times the number of subdevices (up to a maximum of 830 opens). |
| $RECEIVE | One open per process is permitted. |
| Other | Varies by subsystem. |

Specifying a *nowait_depth* value greater than 0 (zero) causes all I/O operations to be performed in a nowait manner. Nowait I/O operations must be completed by a call to the AWAITIOX procedure.

Nowait I/O operations on different file numbers (even if for the same file) are independent, might arrive in any order at the destination, and might be completed by AWAITIOX in any order.

Nowait opens

If you open a file in a nowait manner (*options* bit 1 = 1) and if **SPT_FILE_OPEN_( )** returns no error (returns a value of 0 [zero]), the open operation must be completed by a call to AWAITIOX.

If there is an error, no system message is sent to the object being opened and you do not need to call AWAITIOX to complete the operation. If there is no error, the *filenum* parameter returned by **SPT_FILE_OPEN_( )** is valid; however, you cannot initiate any I/O operation on the file until you complete the open by calling AWAITIOX.

If you specify the *tag* parameter in the call to AWAITIOX, a -30D is returned; the values returned in the *buffer* and *count* parameters to AWAITIOX are undefined. If an error returns from AWAITIOX, it is your responsibility to close the file.

For the TMF transaction pseudofile, or for a waited file (*nowait_depth* = 0 [zero]), a request for a nowait open is rejected.

The Guardian file system implementation of a nowait open might use waited calls in some cases. However, it is guaranteed that the open message is sent using nowait I/O to a process; the opener does not wait for the process being opened to service the open message.

Direct and buffered I/O transfers

A file opened by **SPT_FILE_OPEN_( )** uses direct I/O transfers by default; SETMODE 72 is used to force the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers. This behavior is unlike the obsolescent Guardian OPEN procedure call, which uses a PFS buffer for I/O transfers by default.

Sequential block buffering

Sequential block buffering is only supported for disk files. If you are using sequential block buffering, the file should usually be opened with protected or exclusive access. You can use shared access, but it is somewhat slower than the other access methods, and there might be concurrency problems. See the discussion of "Sequential Block Buffering" in the *Enscribe Programmer’s Guide*.

Named processes

If you supply a process filename for a named process, it can represent any process with the same name. System messages are normally sent to the current primary process. The exception is when a named process supplies its own name to **SPT_FILE_OPEN_( )**. In that case, the name refers to the backup process and system messages are sent to the backup process.

A named process can be represented with or without a sequence number. **SPT_FILE_OPEN_( )** treats the two name forms differently:

- If you supply a process file name that includes a sequence number, the process must have a matching sequence number or the open fails with error 14. When retrying I/O on a process opened under such a name, the file system does not attempt to send messages to a possible backup process of the same name unless it has a matching sequence number. This behavior ensures that the named process is a true backup of the primary process.
- If you supply a process file name that does not include a sequence number, any process with a matching name can be opened and can be sent I/O retries. A newly created process that receives an I/O retry intended for another process of the same name will usually reject it with an error 60, but this behavior is under the control of the application.

Partitioned files

A separate FCB exists for each partition of a partitioned file. There is one ACB per accessor (as for single-volume files), but this ACB requires more main memory because it contains the information necessary to access all of the partitions, including the location and partial-key value for each partition.

Disk file open security check

When a disk file open is attempted, the system performs a security check. The accessor’s (that is, the caller’s) security level is checked against the file security level for the requested access mode, as follows:

for read access read security level is checked.

for write access
write security level is checked.

for read-write access
read and write security levels are checked.

A Guardian file has one of seven levels of security for each access mode. The owner of the file can set the security level for each access mode by using SETMODE function 1 or by using the File Utility Program (FUP) SECURE command. The following table shows the seven levels of security:

Table 7–3. Levels of Guardian File Security

| FUP Code | Program Value | Access Permitted |
|---|---|---|
| - | 7 | Local super ID only |
| U | 6 | Owner (local or remote), that is, any user with owner's ID |
| C | 5 | Member of owner's group (local or remote), that is, any member of owner's community |
| N | 4 | Any user (local or remote) |
| O | 2 | Owner only (local) |
| G | 1 | Member of owner's group (local) |
| A | 0 | Any user (local) |

For a given access mode, the accessor's security level is checked against the file security level. File access is allowed or not allowed as shown in the following table. In this table, file security levels are indicated by FUP security codes. For a given accessor security level, a Y indicates that access is allowed to a file with the security level shown; an X indicates that access is not allowed.

Table 7–4. Allowed Guardian File Accesses

| Accessor's Security Level | File Security Level - | U | C | N | O | G | A |
|---|---|---|---|---|---|---|---|
| Super ID user, local access | Y | Y | Y | Y | Y | Y | Y |
| Super ID user, remote access | X | Y | Y | Y | X | X | X |
| Owner or owner's group manager, remote access | X | Y | Y | Y | X | X | X |
| Member of owner's group, remote access | X | X | Y | Y | X | X | X |
| Any other user, remote access | X | X | X | Y | X | X | X |
| Owner or owner's group manager, local access | X | Y | Y | Y | Y | Y | Y |
| Member of owner's group, local access | X | X | Y | Y | X | Y | Y |

Any other user, local access X X X Y X X Y

If the caller to **SPT_FILE_OPEN_( )** fails the security check, the open fails with an error 48. You can obtain the security level of a file by a call to the Guardian FILE_GETINFOLIST[BYNAME]_ procedure, the FILEINFO procedure, or by the File Utility Program (FUP) INFO command.

If you are using the Safeguard product, this security information might not apply.

Tape file open access mode

The file system does not enforce read-only or write-only access for unlabeled tape, even though no error is returned if you specify one of these access modes when opening a tape file.

File open exclusion and access mode checking

When a file open is attempted, the requested access and exclusion modes are compared with those of any opens already granted for the file. If the attempted open is in conflict with other opens, the open fails with error 12. For a table that lists the possible current modes and requested modes, indicating whether an open succeeds or fails, see the description of the FILE_OPEN_ procedure in the *Guardian Procedure Calls Reference Manual*. For the Optical Storage Facility only, the "process exclusive" exclusion mode is also supported. Process exclusive is the same as exclusive for opens by other processes, but the same as shared for opens by the same process.

Protected exclusion mode

Protected exclusion mode has meaning only for disk files. For other files, specifying protected exclusion mode is equivalent to specifying shared exclusion mode.

**Disk File Considerations**

Maximum number of concurrent nowait operations

The maximum number of concurrent nowait operations permitted for an open of a disk file is 1. Attempting to open a disk file and specify a *nowait_depth* value greater than 1 causes **SPT_FILE_OPEN_( )** to fail with an error 28.

Unstructured files

File pointers after an open

After a disk file is opened, the current-record and next-record pointers begin at a relative byte address (RBA) of 0, and the first data transfer (unless positioning is performed) is from that location. After a successful open, the pointers are:

current-record pointer = 0D
next-record pointer = 0D

Sharing the same EOF pointer

If a given disk file is opened more than once by the same process, separate current-record and next-record pointers are provided for each open, but all opens share the same EOF pointer.

Structured files

Accessing structured files as unstructured files

The unstructured access option (*options* bit 0 = 1) permits a file to be accessed as an unstructured file. You must maintain the block format used by Enscribe if the file is be accessed again in its structured form. (HP reserves the right to change this block format at any time.) For information about Enscribe block formats, see the *Enscribe Programmer's Guide*.

For a file opened using the unstructured access option, a data transfer occurs to the position in the file specified by an RBA (instead of to the position indicated by a key address field or record number); the number of bytes transferred is that specified in the file-system procedure call (instead of the number of bytes indicated by the record format).

If a partitioned file, either structured or unstructured, is opened using the unstructured access option, only the first partition is opened. You must open the remaining partitions individually with separate calls to **SPT_FILE_OPEN_()** (each call specifying unstructured access).

Accessing audited structured files as unstructured files is not allowed.

Current-state indicators after an open

After successful completion of an open, the current-state indicators have these values:

- The current position is that of the first record in the file by primary key.
- The positioning mode is approximate.
- The comparison length is 0.

If the Guardian READ procedure is called immediately after **SPT_FILE_OPEN_()** for a structured file, READ reads the first record in the file; in a key-sequenced file, this is the first record by primary key. Subsequent reads, without intervening positioning, read the file sequentially (in a relative or entry-sequenced file) or by primary key (in a key-sequenced file) through the last record in the file. When a key-sequenced file is opened, the Guardian KEYPOSITION procedure is usually called before any subsequent Guardian I/O procedure call (such as READ, READUPDATE, or WRITE) to establish a position in the file.

Queue files

If the READUPDATELOCK operation is to be used, the value of the *sync_or_receive_depth* parameter must be 0 (zero). You can use a separate open for operations with *sync_or_receive_depth* greater than 0 (zero).

You cannot use sequential block buffering.

64-bit primary keys

In order to access non-key-sequenced files bigger than 4 gigabytes, you must set bit 31 of the **SPT_FILE_OPEN_()** *elections* parameter. Use of this parameter allows the use of procedures using 32-bit primary keys (POSITION, KEYPOSITION, REPOSITION, GETSYNCINFO, and SETSYNCINFO) and the 32-bit key items of the FILE_GETINFOLIST_, FILEINFO, and FILERECINFO procedures.

**Considerations for Terminals**

The terminal used as the operator console should not be opened with exclusive access. If it is, console messages are not logged.

**Interprocess Communication Considerations**

Maximum concurrent nowait operations for an open of $RECEIVE

The maximum number of concurrent nowait operations permitted for an open of $RECEIVE is 1. Attempting to open $RECEIVE and to specify a value greater than 1 causes an error 28 to be returned.

When **SPT_FILE_OPEN_( )** completes

When process A attempts to open process B, **SPT_FILE_OPEN_( )** completes as follows:

- If process B has already opened $RECEIVE with file-management system messages disabled, the open call by process A completes immediately.
- If process B has opened $RECEIVE requesting file-management system messages enabled, the open call completes when process B reads the open message from process A by using READX, or if B uses READUPDATEX, the open call completes when process B replies to the open message (by using REPLYX).

  If process B has not yet opened $RECEIVE, the open by process A does not complete until process B opens $RECEIVE. Specifically, the open by process A completes as follows:

  — When process B opens $RECEIVE with file-management system messages disabled, a waited open by process A completes immediately, but a nowait open by process A completes after the first read of $RECEIVE by process B.

  — When process B opens $RECEIVE with file-management system messages enabled, the open call by process A completes when process B reads the open message from A by using READ[X], or if B uses READUPDATE[X], the open call completes when process B replies to the open message (by using REPLY[X]).

Message formats

When $RECEIVE is opened by **SPT_FILE_OPEN_( )**, system messages are delivered to the caller in D-series format unless messages in C-series format are requested by setting *options* bit 14 to 1. (No file-management system messages are delivered to the caller if *options* bit 15 is set to 1 when opening $RECEIVE.)

Messages from high-PIN processes

Opening $RECEIVE with **SPT_FILE_OPEN_( )** implies that the caller is capable of handling messages from processes with PINs greater than 255.

Opening $RECEIVE and being opened by a remote long-named process

A process that has a process name consisting of more than five characters will fail with an error 20 if it attempts to open a process on a remote node and the process it attempts to open:

- Used the **SPT_FILE_OPEN_( )** procedure to open $RECEIVE and requested that C-series format messages be delivered, or

- Used the Guardian OPEN procedure to open $RECEIVE.

Notification of this failure is not sent to the process reading $RECEIVE.

Opening an unconverted (C-series format) process from a high-PIN process
A high-PIN process cannot open an unconverted process unless the unconverted process has the HIGHREQUESTERS object-file attribute set. If a high-PIN process attempts to open a low-PIN process that does not have this attribute set, the high-PIN process receives file-system error 560.

**System Message**

When a process is opened by either **SPT_FILE_OPEN_( )** or the Guardian OPEN procedure, it receives a process open message (unless it specified when opening $RECEIVE that it wants no messages). This message is in D-series format (message -103) or in C-series format (message -30), depending on what the receiving process specified when it opened $RECEIVE. This message is also received if the backup process of a process pair performs an open. Therefore, a process can expect two of these messages when being opened by a process pair.

You can obtain he process handle of the opener by a subsequent call to FILE_GETRECEIVEINFO_. For a description of the process open message see the *Guardian Procedure Errors and Messages Manual*.

**DEFINE Considerations**

- The *filename* or *pathname* parameter can be a DEFINE name; **SPT_FILE_OPEN_( )** uses the file name given by the DEFINE as the name of the object to be opened. If you specify a CLASS TAPE DEFINE without the DEVICE attribute, the system selects the tape drive to be opened. A CLASS TAPE DEFINE has other effects when supplied to **SPT_FILE_OPEN_( )**. For more information about DEFINEs, see Appendix E of the *Guardian Procedure Calls Reference Manual*.

- If a supplied DEFINE name is a valid name but no such DEFINE exists, the procedure returns an error 198 (missing DEFINE).

- When performing a backup open of a file originally opened with a DEFINE, *filename* must contain the same DEFINE name. The DEFINE must exist and must have the same value as when the primary open was performed.

**Safeguard Considerations**

For information on files protected by Safeguard, see the *Safeguard Reference Manual*.

**OSS Considerations**

- To open an OSS file by its pathname, set *options* bit 10 to 1 and specify the *pathname* parameter.

- You can open OSS files only with shared exclusion mode.

## EXAMPLES

The open in the following example has the following defaults: waited I/O, exclusion mode (shared), access mode (read/write), sync depth (0).

**error = SPT_FILE_OPEN_ ( filename, filenum );**

## RETURN VALUES

The **SPT_FILE_OPEN_( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

This function can return any error number that the Guardian FILE_OPEN_ procedure call can return. It can also return the following error number:

| | |
|---|---|
| 12 | Callback has already been registered for this *filenum*. |

Some error numbers are warnings (that is, they indicate conditions that do not prevent the file from being opened); check the value returned for the *filenum* parameter to determine whether the file was opened successfully. Forexplanation of other error numbers returned, see the *Guardian Procedure Errors and Messages Manual*.

### ERRORS

None. This function does not set the **errno** variable.

### RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

### STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fork -** Initiates a thread-aware fork() operation

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**pid_t spt_fork(void);**

## PARAMETERS

None.

## DESCRIPTION

This is a thread-aware version of the **fork( )** function call that creates a new process from the current thread.

The following macro maps the **spt_fork( )** call to the **fork( )** funciton and has been defined in the **spthread.h** header file:

**#define fork() spt_fork().**

## NOTES

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll*nnn*/zsptdll**).

## RETURN VALUES

See the **fork(2)** reference page.

## RELATED INFORMATION

Functions: **fork(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fprintf -** Initiates thread-aware **fprintf( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <spthread.h>

int spt_fprintf(
        FILE *stream,
        const char *format,
         ...);
```

## PARAMETERS

See the **fprintf(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **fprintf( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread aware.

The following macro maps **spt_fprintf( )** to **fprintf( )** and has been defined in **spthread.h**:

**#define fprintf spt_fprintf**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **fprintf(3)** reference page. The following also applies:

- Value **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying the stream becomes invalid (is closed by another thread), **-**1 is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **-**1 is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fprintfx -** Prints formatted output to an output stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

**int spt_fprintfx (**
**FILE** ***stream*,**
**const char** ***format*
[**,** *value*] . . .
**);**

## PARAMETERS

*stream* Points to a **FILE** structure specifying an open stream to which converted values will be written.

*format* Specifies a character string combining literal characters with conversion specifications.

*value* Specifies the data to be converted according to the *format* parameter.

## DESCRIPTION

The **spt_fprintfx( )** function is the thread-aware version of the **fprintf( )** function.

The **spt_fprintfx( )** function converts, formats, and writes its *value* parameters, under control of the *format* parameter, to the output stream specified by its *stream* parameter.

The *format* parameter is a character string that contains two types of objects:

- Literal characters, which are copied to the output stream.
- Conversion specifications, each of which causes zero or more items to be fetched from the *value* parameter list.

If not enough items for *format* are in the *value* parameter list, the results are unpredictable. If more *value*s remain after the entire *format* has been processed, they are ignored.

### Conversion Specifications

Each conversion specification in the *format* parameter has the following syntax:

- A **%** (percent sign).

  The **spt_fprintfx( )** function can handle a format string that enables the system to process elements of the parameter list in variable order. In such a case, the normal conversion character **%** (percent sign) is replaced by **%***digit***$**, where *digit* is a decimal number in the range from 1 to **NL_ARGMAX**. Conversion is then applied to the specified argument, rather than to the next unused argument. This feature provides for the definition of format strings in an order appropriate to specific languages. When variable ordering is used, the * (asterisk) specification for field width in precision is replaced by **%***digit***$**. If the variable ordering feature is used, it must be specified for all conversions.

- Zero or more flags that modify the meaning of the conversion specification. The flag characters and their meanings are:

| | |
|---|---|
| **-** | Left align the result of the conversion within the field. |
| **+** | Begin the result of a signed conversion with a sign (+ or -). |
| (space) | Prefix a space character to the result if the first character of a signed conversion is not a sign. If both the (space) and **+** flags appear, the (space) flag is ignored. |
| **#** | Convert the value to an alternate form. For **o** conversion, it increases the precision to force the first digit of the result to be a 0 (zero). For **x** and **X** conversions, a nonzero result has 0x or 0X prefixed to it. For **e, E, f, g,** and **G** conversions, the result always contains a radix character, even if no digits follow it. For **g** and **G** conversions, trailing zeros are not removed from the result. For **c**, **C**, **d**, **i**, **s**, **S**, and **u** conversions, the flag has no effect. |
| **0** | Pad to field width using leading zeros (following any indication of sign or base) for **d**, **e**, **E**, **f**, **g**, **G**, **i**, **o**, **u**, **x**, and **X** conversions; no space padding is performed. If the **0** and **-** (dash) flags both appear, the **0** flag will be ignored. For **d**, **i**, **o u**, **x**, and **X** conversions, if a precision is specified, the **0** flag is also ignored. For other conversions, the behavior is undefined. |

- An optional decimal digit string that specifies the minimum field width. If the converted value has fewer characters than the field width, the field is padded on the left to the length specified by the field width. If the left-adjustment flag is specified, the field is padded on the right.

  A field width can be indicated by an * (asterisk) instead of a digit string. In this case, an integer (**int**) *value* parameter supplies the field width. The *value* parameter converted for output is not fetched until the conversion letter is reached, so the parameters specifying field width or precision must appear before the value (if any) to be converted. If the corresponding parameter has a negative value, it is treated as a **-** (dash) left alignment option followed by a positive field width. When variable ordering with the **L***digit***$** format is used, the ***** (asterisk) specification for field width in precision is replaced by ******digit***$**.

- An optional precision. The precision is a **.** (dot) followed by a decimal digit string. If no precision is given, it is treated as 0 (zero). The precision specifies:

  — The minimum number of digits to appear for the **d**, **u**, **o**, **x**, or **X** conversions.

  — The number of digits to appear after the radix character for the **e**, **E**, and **f** conversions.

  — The maximum number of significant digits for the **g** and **G** conversions.

  — The maximum number of bytes to be printed from a string in the **s** or **S** conversion.

  A field precision can be indicated by an * (asterisk) instead of a digit string. In this case, an integer (**int**) *value* parameter supplies the field precision. The *value* parameter converted for output is not fetched until the conversion letter is reached, so the parameters specifying field width or precision must appear before the value (if any) to be converted. If the value of the corresponding parameter is negative, it is treated as if the precision had not been specified. When variable ordering with the **L***digit***$** format is used, the ***** (asterisk) specification for field width in precision is replaced by ******digit***$**.

- An optional **h**, **l**, **ll**, or **L** indicating the size of the argument corresponding to the following integer or floating-point conversion specifier:

  — An **h** followed by a **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier indicates that the argument will be treated as a **short int** or **unsigned short int**.

  — An **h** followed by an **n** conversion specifier indicates that the argument will be treated as a pointer to a **short int**.

  — An **l** followed by a **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier indicates that the argument will be treated as a **long int** or **unsigned long int**.

  — An **l** followed by an **n** conversion specifier indicates that the argument will be treated as a pointer to a **long int**.

  — An **ll** followed by a **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier indicates that the argument will be treated as a **long long int** or **unsigned long long int**.

  — An **ll** followed by an **n** conversion specifier indicates that the argument will be treated as a pointer to a **long long int**.

  — An **L** followed by a **e**, **E**, **f**, **g**, or **G** conversion specifier indicates that the argument will be treated as a **long double**.

  — An **L** followed by a **d**, **i**, **o**, **x**, or **X** conversion specifier indicates that the argument will be treated as a **long long**, which is a 64-bit integer data type and an HP extension.

- A character that indicates the type of conversion to be applied:

**%** Performs no conversion. Prints **%**.

**d** or **i** Accepts an integer (**int**) *value* and converts it to signed decimal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a 0 (zero) value with a precision of 0 (zero) is a null string. Specifying a field width with a 0 (zero) as a leading character causes the field width value to be padded with leading zeros.

**u** Accepts an integer (**int**) *value* and converts it to unsigned decimal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a 0 (zero) value with a precision of 0 (zero) is a null string. Specifying a field width with a 0 (zero) as a leading character causes the field width value to be padded with leading zeros.

**o** Accepts an integer (**int**) *value* and converts it to unsigned octal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a 0 (zero) value with a precision of 0 (zero) is a null string. Specifying a field width with a 0 (zero) as a leading character causes the field width value to be padded with leading zeros. An octal value for field width is not implied.

**x** or **X**
: Accepts an integer (**int**) *value* and converts it to unsigned hexadecimal notation. The letters abcdef are used for the **x** conversion and the letters ABCDEF are used for the **X** conversion. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a 0 (zero) value with a precision of 0 (zero) is a null string. Specifying a field width with a 0 (zero) as a leading character causes the field width value to be padded with leading zeros.

**f**
: Accepts a **float** or **double** *value* and converts it to decimal notation in the format [-]*ddd.ddd*. The number of digits after the radix character is equal to the precision specification. If no precision is specified, six digits are output. If the precision is 0 (zero), no radix character appears (unless the **#** flag is specified). If a radix character is output, at least one digit is output before it. The value is rounded to the appropriate number of digits.

**e** or **E**
: Accepts a **float** or **double** *value* and converts it to the exponential form [-]*d.ddd***e**+/-*dd*. One digit is before the radix character and the number of digits after the readix character is equal to the precision specification. If no precision is specified, six digits are output. If the precision is 0 (zero), no radix character appears (unless the **#** flag is specified). The **E** conversion character produces a number with uppercase **E** instead of lowercase **e** before the exponent. The exponent always contains at least two digits. If the value is 0 (zero), the exponent is 0 (zero).

**g** or **G**
: Accepts a **float** or **double** *value* and converts it in the style of the **e**, **E**, or **f** conversion characters, with the precision specifying the number of significant digits. Trailing zeros are removed from the result. A radix character appears only if it is followed by a digit (except that it always appears if the **#** flag is specified). The style used depends on the value converted. Style **e** (**E**, if **G** is the flag used) results only if the exponent resulting from the conversion is less than -4, or if it is greater or equal to the precision.

**c**
: Accepts and prints an integer (**int**) *value* converted to an **unsigned char**.

**C**
: Accepts a **wchar_t** value, converts it to an array of bytes containing a multibyte character, and prints it. If a minimum field width is specified and the multibyte character occupies fewer bytes than the specified width, the multibyte character is padded with space characters to the specified width.

**s**
: Accepts a pointer to an array of **char** type. Bytes from the array are printed until a null character is encountered or the number of characters indicated by the precision is reached. If no precision is specified, all characters up to the first null character are printed. If the precision is not specified or is greater than the size of the array, the array must be terminated by a null byte. If the string pointer *value* has a value of 0 (zero) or null, the results are undefined.

**S** Accepts a pointer to an array of **wchar_t** type. Wide characters from the array are converted to an array of bytes containing multibyte characters and the multibyte characters up to (but not including) the null character are printed. If a precision is specified, no more than the number of bytes specified by the precision are printed. If the precision is not specified or is greater than the size of the array of bytes, the array of wide characters must be terminated by a null wide character. If a minimum field width is specified and the array of bytes occupy fewer bytes than the specified width, the array is padded with space characters to the specified width.

**p** Accepts a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, the same as unsigned hexadecimal integer (**x**).

**n** Accepts a pointer to an integer into which is written the number of characters written to the output stream so far by this call. No argument is converted.

If the result of a conversion is wider than the field width, the field is expanded to contain the converted result. No truncation occurs. However, a small precision can cause truncation on the right.

The **e**, **E**, **f**, and **g** formats represent the special floating-point values as follows:

| | |
|---|---|
| Quiet NaN | **NaN** |
| Signaling NaN | **NaN** |
| +/-INF | **+Inf** or **-Inf** |
| +/-0 | +0.0 or -0.0 (zero) |

The representation of the **+** (plus sign) depends on whether the **+** or (space) formatting flag is specified.

The **spt_fprintfx( )** function allows for the insertion of a language-dependent radix character in the output string. The radix character is defined by **langinfo** data in the program's locale (category **LC_NUMERIC**). In the C locale, or in a locale where the radix character is not defined, the radix character defaults to **.** (period).

The **st_ctime** and **st_mtime** fields of the file are marked for update between the successful execution of the **spt_fprintfx( )** function and the next successful completion of a call to the **spt_fflushx( )** or **spt_fclosex( )** functions on the same stream, or a call to the **exit( )** or **abort( )** functions.

## NOTES

The macro to map **fprintf( )** to **spt_fprintfx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **fprint( )** to **spt_fprintfx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

This function supports both IEEE Std 754-1985 floating-point and Tandem floating-point values in the native environment. IEEE values can include NaN and infinity, and the sign of 0.0 (zero) can be either positive or negative. For a description of the IEEE value classes, see the **fp_class(3)** reference page.

Guardian functions are available to convert between floating-point formats. For a discussion of floating-point conversions, see the *Guardian Programmer's Guide*.

## RETURN VALUES

Upon successful completion, this function returns the number of bytes in the output string. Otherwise, a negative value is returned.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## ERRORS

The **spt_fprintfx( )** function fails if either:

- *stream* is unbuffered
- The buffer for *stream* needed to be flushed and the function call caused an underlying **spt_writex( )** or **lseek( )** function to be invoked.

In addition, if the **spt_fprintfx( )** function fails, **errno** is set to one of the following values:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EILSEQ] An invalid wide character was detected.

[EINTR] The read operation was interrupted by a signal that was caught, and no data was transferred.

[EINVAL] There are insufficient arguments.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned. This error might also be returned under implementation-defined conditions.

[ENOMEM] Insufficient storage space was available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

**RELATED INFORMATION**

Functions: **fp_class(3)**, **fprintf(3)**, **isnan(3)**, **toascii(3)**, **printf(3)**, **putc(3)**, **scanf(3)**, **sprintf(3)**, **spt_fprintf(2)**, **spt_printfx(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fputc -** Thread-aware **fputc( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_fputc(**
**int** *c***,**
**FILE ****stream***);**

## PARAMETERS

See the **fputc(3)** man page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **fputc( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread aware.

The following macro maps **spt_fputc( )** to **fputc( )** and has been defined in **spthread.h**:

**#define fputc(c, stream) spt_fputc(c, stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **fputc(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying the stream becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function that is not blocked, ignored, or handled, EOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fputcx -** Writes a byte to a specified output stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

**int spt_fputcx (**
**int** *c*,
**FILE ****stream*
**);**

## PARAMETERS

*c* Specifies the character to be written.

*stream* Points to the file structure of an open file.

## DESCRIPTION

The **spt_fputcx( )** function is the thread-aware version of the **fputc( )** function.

The **spt_fputcx( )** function writes the character *c* to the output specified by the *stream* parameter. The character is written at the position at which the file pointer is currently pointing, if defined.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen( )** function causes it to become buffered or line buffered. Use the **setbuf( )** function to change the stream-buffering strategy.

When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as it is written. When an output stream is buffered, many characters are saved and written as a block. When an output stream is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a newline character is written or terminal input is requested).

The **st_ctime** and **st_mtime** fields of the file are marked for update between the successful execution of the **spt_fputcx( )** function, and the next successful completion of a call to the **spt_fflushx( )** or **spt_fclosex( )** function on the same stream, or a call to the **exit( )** or **abort( )** function.

## NOTES

The macro to map **fputc( )** to **spt_fputcx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **fputc( )** to **spt_fputcx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

The **spt_fputcx( )** function is never a macro.

The **spt_fputcx( )** function runs more slowly than **spt_putcx( )**, but takes less space per invocation.

## RETURN VALUES

The **spt_fputcx( )** function, upon successful completion, returns the value written. If this function fails, it returns the constant **EOF** and sets **errno**.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), **EOF** is returned with an **errno** value of [EBADF]. If a signal is received via **pthread_kill(2)** and is not blocked, ignored, or handled, **EOF** is returned with an **errno** value of [EINTR].

## ERRORS

The **spt_fputcx( )** function fails if:

- The *stream* parameter is not open for writing.
- The output file size cannot be increased.
- The *stream* is unbuffered.
- The buffer of the *stream* needs to be flushed and the function call causes an underlying **spt_writex( )** or **lseek( )** to be invoked, and this underlying operation fails.

In addition, if any of these conditions occur, the **spt_fputcx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying the output stream and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying the output stream is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EINTR] The write operation was interrupted by a signal that was caught, and no data was transferred.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned. This error might also be returned under implementation-defined conditions.

[ENOMEM] Insufficient memory storage space is available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

Any error encountered during the underlying call to the **spt_writex( )** function can cause this function to return the corresponding **errno** value reported by the **spt_writex( )** function. If your application program encounters an **errno** value not listed on this reference page, refer to the **spt_writex(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the cause of that error.

## RELATED INFORMATION

Functions: **ferror(3)**, **fputc(3)**, **getc(3)**, **getwc(3)**, **printf(3)**, **putc(3)**, **putchar(3)**, **puts(3)**, **putwc(3)**, **spt_getcx(2)**, **spt_getwcx(2)**, **spt_printfx(2)**, **spt_putcx(2)**, **spt_putcharx(2)**, **spt_putsx(2)**, **spt_putwcx(2)**, **spt_writex(2)**, **write(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fputs** - Initiates thread-aware **fputs( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_fputs(**
    **const char ****string***,**
    **FILE ****stream***);**

## PARAMETERS

See the **fputs(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **fputs( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread aware.

The following macro maps **spt_fputs( )** to **fputs( )** and has been defined in **spthread.h**:

**#define fputs(string, stream) spt_fputs(string, stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **fputs(3)** reference page. The following information also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying stream becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fputsx -** Writes a string to a stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

**int spt_fputsx (**
    **const char ****string***,**
    **FILE ****stream*
    **);**

## PARAMETERS

*string* Points to a string to be written to output.

*stream* Points to the **FILE** structure of an open file.

## DESCRIPTION

The **spt_fputsx( )** function is the thread-aware version of the **fputs( )** function.

The **spt_fputsx( )** function writes the null-terminated string pointed to by the *string* parameter to the output stream specified by the *stream* parameter. The **spt_fputsx( )** function does not append a newline character or write the terminating null byte.

The **st_ctime** and **st_mtime** fields of the file are marked for update between the successful execution of the **spt_fputsx( )** function, and the next successful completion of a call to the **spt_fflush( )** or **spt_fclose( )** function on the same stream, or a call to the **exit( )** or **abort( )** function.

## NOTES

The macro to map **fputs( )** to **spt_fputsx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **fputs( )** to **spt_fputsx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

Upon successful completion, the **spt_fputsx( )** function returns the number of characters written. This function can return **EOF** on an error.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), **EOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function that is not blocked, ignored, or handled, **EOF** is returned with an **errno** value of [EINTR].

## ERRORS

The **spt_fputsx( )** function fails if either:

- The *stream* is unbuffered
- The buffer of the *stream* needs to be flushed and the function call causes an underlying **spt_writex( )** or **lseek( )** to be invoked and this underlying operation fails with

incomplete output.

In addition, if any of these conditions occur, the **spt_fputsx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EINTR] The operation was interrupted by a signal that was caught, and no data was transferred.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**, and the process group of the process is orphaned. This error might also be returned under implementation-defined conditions.

[ENOMEM] Insufficient storage space available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

## RELATED INFORMATION

Functions: **fputs(3)**, **gets(3)**, **getws(3)**, **printf(3)**, **putc(3)**, **puts(3)**, **putwc(3)**, **putws(3)**, **spt_getsx(2)**, **spt_getwsx(2)**, **spt_fprintfx(2)**, **spt_putcx(2)**, **spt_putsx(2)**, **spt_putwcx(2)**, **spt_putwx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fputwc** - Thread-aware **fputwc( )**

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**wint_t spt_fputwc(**
    **wint_t** *c***,**
    **FILE ****stream***);**

## PARAMETERS

See **fputwc(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **fputwc( )** function. The file descriptor underlying the *stream* must be nonblocking for this function to be thread aware.

The following macro maps **spt_fputwc( )** to **fputwc( )** and has been defined in **spthread.h**:

**#define fputwc(c, stream) spt_fputwc(c, stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **fputwc(3)** reference page. The following information also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying the *stream* becomes invalid (is closed by another thread), WEOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, WEOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fputwcx -** Writes a wide character to a specified stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <wchar.h>**]
**#include <spthread.h>**

**wint_t spt_fputwcx (**
  **wint_t** *c***,**
  **FILE** *****stream*
  **);**

## PARAMETERS

*c* Specifies the wide character to be written.

*stream* Points to the output data.

## DESCRIPTION

The **spt_fputwcx( )** function is the thread-aware version of the **fputwc( )** function.

The **spt_fputwcx( )** function converts the **wchar_t** specified by the *c* parameter to its equivalent multibyte character and then writes the multibyte character to the *stream* parameter.

The **spt_fputwcx( )** function works the same as **spt_putwcx( )**.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen( )** function causes it to become buffered or line buffered. Use the **setbuf( )** function to change the stream's buffering strategy.

## NOTES

The macro to map **fputwc( )** to **spt_fputwcx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **fputwc( )** to **spt_fputwcx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

Upon successful completion, this function returns the value written. If this function fails, it returns the constant **WEOF** (wide-character end-of-file).

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), **WEOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **WEOF** is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_fputwcx( )** function sets **errno** to the corresponding value.

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EINTR] The operation was interrupted by a signal that was caught, and no data was transferred.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned.

[ENOMEM] Insufficient storage space is available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

[EILSEQ] The wide character code specified by the *c* parameter does not correspond to a valid character.

## RELATED INFORMATION

Functions: **fputc(3)**, **fputwc(3)**, **getc(3)**, **getwc(3)**, **printf(3)**, **putc(3)**, **puts(3)**, **putwc(3)**, **putwchar(3)**, **spt_fputwc(2)**, **spt_fputcx(2)**, **spt_getcx(2)**, **spt_getwcx(2)**, **spt_fprintfx(2)**, **spt_putcx(2)**, **spt_putsx(2)**, **spt_putwcx(2)**, **spt_putwcharx(2)**, **wctomb(3)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fread -** Initiates thread-aware **fread( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**size_t spt_fread(**
    **void** *****pointer**,**
    **size_t** *size***,**
    **size_t** *num_items***,**
    **FILE** *****stream);

## PARAMETERS

See the **fread(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **fread( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread aware.

The following macro maps **spt_fread( )** to **fread( )** and has been defined in **spthread.h**:

```
#define fread(pointer, size, num_items, stream)
        spt_fread(pointer, size, num_items, stream)
```

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE
```

## RETURN VALUES

See the **fread(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying stream becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, 0 is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_freadx -** Reads input from a stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

**size_t spt_freadx (**
**void** *****pointer***,**
**size_t** *size***,**
**size_t** *num_items***,**
**FILE** *****stream*
**);**

## PARAMETERS

*pointer* Points to an array.

*size* Specifies the size of the variable type of the array pointed to by the *pointer* parameter.

*num_items* Specifies the number of items of data.

*stream* Specifies the input output stream.

## DESCRIPTION

The **spt_freadx( )** function is the thread-aware version of the **fread( )** function.

The **spt_freadx( )** function copies *num_items* of data of length *size* from the input stream into an array beginning at the location pointed to by the *pointer* parameter.

The **spt_freadx( )** function stops copying bytes if an end-of-file or error condition is encountered while reading from the input specified by the *stream* parameter, or when the number of data items specified by the *num_items* parameter have been copied. It leaves the file pointer of the *stream* parameter, if defined, pointing to the byte following the last byte read, if there is one. The **fspt_readx( )** function does not change the contents of the *stream* parameter.

## NOTES

The macro to map **fread( )** to **spt_freadx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **fread( )** to **spt_freadx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

Upon successful completion, the **spt_freadx( )** function returns the number of items actually transferred. If the *num_items* parameter is negative or 0 (zero), no characters are transferred, and a value of 0 (zero) is returned. If a read error occurs, the error indicator for the stream is set, and **errno** is set to indicate the error.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), **EOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **EOF** is returned with an **errno** value of [EINTR].

## ERRORS

The **spt_freadx( )** function fails if:

- The *stream* parameter is not open for reading.
- The *stream* is unbuffered.
- The *stream*'s buffer needed to be flushed and the function call caused an underlying **spt_writex( )** or **lseek( )** to be invoked and this underlying operation fails.

In addition, if any of the following conditions occur, the **spt_freadx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying the input stream and the process would be delayed in the read operation.

[EBADF] The file descriptor underlying the input stream is not a valid file descriptor open for reading.

[EINTR] The read operation was interrupted by a signal that was caught, and no data was transferred.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.

[ENOMEM] Insufficient memory storage space is available.

[ENOSPC] There was no free space remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to read from a pipe or FIFO that is not open for writing by any process. A **SIGPIPE** signal will also be sent to the process.

Any error encountered during the underlying call to the **spt_readx( )** function can cause this function to return the corresponding **errno** value reported by the **spt_readx( )** function. If your application program encounters an **errno** value not listed above, refer to the **spt_readx(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the cause of that error.

## RELATED INFORMATION

Functions: **fopen(3)**, **fread(3)**, **fwrite(3)**, **getc(3)**, **gets(3)**, **printf(3)**, **putc(3)**, **puts(3)**, **read(2)**, **scanf(3)**, **spt_fwritex(2)**, **spt_getcx(2)**, **spt_getsx(2)**, **spt_printfx(2)**, **spt_putcx(2)**, **spt_putsx(2)**, **spt_readx(2)**, **spt_writex(2)**, **write(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fstat64z -** Provides information about an open file (serializes I/O operations on an open file))

## LIBRARY

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <sys/types.h>** /* optional except for POSIX.1 */
**#include <sys/stat.h>**
**#include <spthread.h>**

**int spt_fstat64z(**
  **int** *filedes***,**
  **struct stat64 ****buffer***);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**,**dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*buffer* Points to a **stat64** structure, into which information is placed about the file. The **stat64** structure is described in the **sys/stat.h** header file.

## DESCRIPTION

The **spt_fstat64z( )** function is a thread-aware version of the **fstat64( )** function.

The **spt_fstat64z( )** function obtains information about the open file associated with the *filedes* parameter.

The file information is written to the area specified by the *buffer* parameter, which is a pointer to a **stat64** structure. For J06.11 and later J-series RVUs and H06.22 and later H-series RVUs, the **stat64** structure uses this definition from the **sys/stat.h** header file:

```
struct stat64 {
        dev_t       st_dev;
        ino64_t     st_ino;
        mode_t      st_mode;
        nlink_t     st_nlink;
        unsigned int        st_acl:1;
        unsigned int        __filler_1:7;
        unsigned int        st_fileprivs:8; /* File privileges */
        uid_t       st_uid;
        gid_t       st_gid;
        dev_t       st_rdev;
        off64_t     st_size;
        time_t      st_atime;
        time_t      st_mtime;
        time_t      st_ctime;
        mode_t      st_basemode; /* Permissions with original group perms */
        int64_t     reserved[3];
};
```

For J06.10 and earlier J-series RVUs and H06.21 and earlier H-series RVUs, the **stat64** structure uses this definition from the **sys/stat.h** header file:

```
struct stat64 {
        dev_t       st_dev;
        ino64_t     st_ino;
        mode_t      st_mode;
```

```
 nlink_t  st_nlink;
          unsigned int      st_acl:1;
          unsigned int      __filler_1:15;
          uid_t     st_uid;
          gid_t     st_gid;
          dev_t     st_rdev;
          off64_t   st_size;
          time_t    st_atime;
          time_t    st_mtime;
          time_t    st_ctime;
          mode_t    st_basemode; /* Permissions with original group perms */
          int64_t   reserved[3];
};
```

The **spt_fstat64z( )** function updates any time-related fields associated with the file before writing into the **stat64** structure, unless it is a read-only fileset. Time-related fields are not updated for read-only OSS filesets.

The fields in the **stat64** structure have these meanings and content:

**st_dev** OSS device identifier for a fileset.

Values for local OSS objects are listed next. Values for local Guardian objects are described in **Use on Guardian Objects**, and values for remote Guardian or OSS objects are described in **Use on Remote Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | ID of device containing directory entry |
| Directory | ID of device containing directory |
| Pipe or FIFO | ID of special fileset for pipes |
| **AF_INET** or **AF_INET6** socket | ID of special fileset for sockets |
| **AF_UNIX** socket | ID of device containing the fileset in which the socket file was created |
| **/dev/null** | ID of device containing directory entry |
| **/dev/tty** | ID of device containing directory entry |

**st_ino** File serial number (inode number). The file serial number and OSS device identifier uniquely identify a regular OSS file within an OSS fileset.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | File serial number (unique) |
| Directory | File serial number (unique) |
| Pipe or FIFO | File serial number (unique) |
| **AF_INET** or **AF_INET6** socket | File serial number (not unique within the HP NonStop node) |
| **AF_UNIX** socket | File serial number of the socket file (unique) |
| **/dev/null** | File serial number (unique) |
| **/dev/tty** | File serial number (unique) |

The **st_ino** value for all node entries in **/E** (including the entry for the logical link from the local node name to the root fileset on the local node) is the value for the root fileset on the corresponding node. If normal conventions are followed, this value is always 0 (zero), so entries in **/E** appear to be nonunique. Values for objects on remote nodes are unique only among the values for objects within the same fileset on that node.

**st_mode** File mode. These bits are ORed into the **st_mode** field:

**S_IFMT** File type. This field can contain one of these values:

**S_IFCHR** Character special file.

**S_IFDIR** Directory.

**S_IFIFO** Pipe or FIFO.

**S_IFREG** Regular file.

**S_IFSOCK** Socket.

For an **AF_INET** or **AF_INET6** socket, the user default permissions are returned for the permission bits. The access flags are set to read and write.

For an **AF_UNIX** socket, the user permissions from the inode for the socket are returned for the permission bits. The access flags are also returned from the inode.

**S_IRWXG** Permissions for the owning group, or if the **st_acl** flag is set, permissions for the the **class** ACL entry.

**S_IRWXO** Other class

**S_IRWXU** Owner class

**S_ISGID** Set group ID on execution

**S_ISUID** Set user ID on execution

**S_ISVTX** Sticky bit; used only for directories (not ORed for files in **/G**, the Guardian file system)

**S_TRUST** Indicates that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers when the memory segment containing the buffers is not shared. This flag applies only to loadfiles for a process, and only a user with appropriate privileges (the super ID) can set it.

**S_TRUSTSHARED**
Indicates that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers regardless of whether the memory segment containing the buffers is shared. This flag applies only to loadfiles for a process, and only a user with appropriate privileges (the super ID) can set it.

Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

**st_nlink** Number of links.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Number of links to the file |
| Directory | Number of links to the directory |
| FIFO | Number of links to the file |
| Pipe | -1 |
| **AF_INET** or **AF_INET6** socket | 0 (zero) |
| **AF_UNIX** socket | Number of links to the socket file |
| **/dev/null** | Number of links to the file |
| **/dev/tty** | Number of links to the file |

**st_acl** If set to 1, indicates that the file has optional access control list (ACL) entries. For compatibility with HP-UX, the member name **st_aclv** is provided as alias for **st_acl**. For more information about ACLs, see the **acl(5)** reference page.

**st_fileprivs** File privileges. For information about file privileges see the **setfilepriv(2)** reference page.

**st_uid** User ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | User ID of the file owner |
| Directory | User ID of the file owner |
| Pipe or FIFO | User ID of the file owner |
| **AF_INET** or **AF_INET6** socket | User ID of the calling process |
| **AF_UNIX** socket | User ID of the creator of the socket file |
| **/dev/null** | User ID of the super ID |
| **/dev/tty** | User ID of the super ID |

**st_gid** Group ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Group ID of the file group |
| Directory | Group ID of the file group |
| Pipe or FIFO | Group ID of the file group |
| **AF_INET** or **AF_INET6** socket | Group ID of the calling process |
| **AF_UNIX** socket | Group ID of the creator of the socket file |
| **/dev/null** | Group ID of the super ID |
| **/dev/tty** | Group ID of the super ID |

**st_basemode** If the **st_acl** flag is set, contains the permissions for the file owner, owning group, and others. If the **st_acl** flag is not set, **st_basemode** is 0 (zero).

**st_rdev** Remote device ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Undefined |
| Directory | Undefined |
| Pipe or FIFO | Undefined |
| **AF_INET** or **AF_INET6** socket | 0 (zero) |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | Undefined |
| **/dev/tty** | ID of the device |

**st_size** File size.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Size of the file in bytes |
| Directory | 4096 |
| Pipe or FIFO | 0 (zero) |
| **AF_INET** or **AF_INET6** socket | 0 (zero) |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | 0 (zero) |
| **/dev/tty** | 0 (zero) |

**st_atime** Access time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last access |
| Directory | Time of the last access |
| Pipe or FIFO | Time of the last access |
| **AF_INET** or **AF_INET6** socket | Value maintained in the socket data structure |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_mtime** Modification time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last data modification |
| Directory | Time of the last modification |
| Pipe or FIFO | Time of the last data modification |
| **AF_INET** or **AF_INET6** socket | Value maintained in the socket data structure |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_ctime** Status change time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last file status change |
| Directory | Time of the last file status change |
| Pipe or FIFO | Time of the last file status change |
| **AF_INET** or **AF_INET6** socket | Value maintained in the socket data structure |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**Use on Guardian Objects**

The **st_dev** and **st_ino** fields of the **stat64** structure do not uniquely identify Guardian files (files in **/G**).

The **st_dev** field is unique for **/G**, for each disk volume, and for each Telserv process (or other process of subdevice type 30), because each of these is a separate fileset.

The **S_ISGUARDIANOBJECT** macro can indicate whether an object is a Guardian object when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is a Guardian object and **FALSE** otherwise.

The **st_ino** field is a nonunique encoding of the Guardian filename.

The **st_rdev** field contains a unique minor device number for each entry in **/G/ztnt/**, representing each Telserv process subdevice.

The **st_size** field of an EDIT file (file code 101) is the actual (physical) end of file, not the number of bytes in the file. For directories, **st_size** is set to 4096.

When an OSS function is called for a Guardian EDIT file, the **st_mtime** field is set to the last modification time. The **st_atime** field indicates the last time the file was opened, and the **st_ctime** field is set equal to **st_mtime**. No other time-related fields are updated by OSS function calls.

The **st_ctime** and **st_atime** fields for Guardian regular disk files (except for EDIT files) are updated by OSS function calls, not by Guardian procedure calls.

The time fields for **/G**, **/G/***vol*, and **/G/***vol***/***subvol* always contain the current time.

The mapping between Guardian files and their corresponding file types described in the **st_mode** field is listed next:

| Example in /G | Guardian File Type | st_mode File Type | Permissions |
|---|---|---|---|
| **/G** | N/A | Directory | r-xr-xr-x |
| *vol* | Disk volume | Directory | rwxrwxrwx |
| *vol*/*subvol* | Subvolume | Directory | rwxrwxrwx |
| *vol*/*subvol*/*fileid* | Disk file | Regular file | See following text |
| *vol*/**#123** | Temporary disk file | Regular file | See following text |
| **ztnt** | Subtype 30 process | Directory | --x--x--x |
| **ztnt/#pty0001** | Subtype 30 process with qualifier | Character special | rw-rw-rw- |
| **vol1/zyq00001** | Subvolume | Directory | --------- |

A Guardian file classified as a directory is always owned by the super ID.

Guardian permissions are mapped as follows:

- Guardian network or any user permission is mapped to OSS other permission.
- Guardian community or group user permission is mapped to OSS group permission.
- Guardian user or owner permission is mapped to OSS owner permission.
- Guardian super ID permission is OSS super ID permission.
- Guardian read permission is mapped to OSS read permission.
- Guardian write permission is mapped to OSS write permission.
- Guardian execute permission is mapped to OSS execute permission.
- Guardian purge permission is ignored.

Users are not allowed read access to Guardian processes.

OSS file permissions are divided into three groups (owner, group, and other) of three permission bits each (read, write, and execute). The OSS permission bits do not distinguish between remote and local users as Guardian security does; local and remote users are treated alike.

### Use on Remote Objects

The content of the **st_dev** field of the **stat64** structure is unique for each node in **/E** because each node is a separate fileset. Values for directories within **/E** are the same as values for objects on the local HP NonStop node.

The **S_ISEXPANDOBJECT** macro can indicate whether an object in the **/E** directory is on a remote HP NonStop server node when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is on a remote HP NonStop node and **FALSE** otherwise.

## NOTES

This function serializes file operations on an open file. If a thread calls **spt_fstat64z( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **fstat( )** to **spt_fstat64z( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** and the **#define _FILE_OFFSET_BITS 64** preprocessor directives before including **spthread.h** or when you use equivalent compiler command options to compile the application.

For C++ applications, an alias to map **fstat( )** to **spt_fstat64z( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** and the **#define _FILE_OFFSET_BITS 64** preprocessor directives before including **spthread.h** or when you use equivalent compiler command options to compile the application.

For C applications, a macro to map **fstat64( )** to **spt_fstat64z( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** and the **#define _LARGEFILE64_SOURCE 1** preprocessor directives before including **spthread.h** or when you use equivalent compiler command options to compile the application.

For C++ applications, an alias to map **fstat64( )** to **spt_fstat64z( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** and the **#define _LARGEFILE64_SOURCE 1** preprocessor directives before including **spthread.h** or when you use equivalent compiler command options to compile the application.

A direct application call to **spt_fstatz( )** is automatically mapped to **spt_fstat64z( )** when you use the **#define _FILE_OFFSET_BITS 64** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **spt_fstat64z( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not a valid file descriptor.

[EFAULT] The *buffer* parameter points to a location outside of the allocated address space of the process.

[EFSBAD] The program attempted an operation involving a fileset with a corrupted fileset catalog.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOROOT] The program attempted an operation while the root fileset was unavailable.

[ENXIO] An invalid device or address was specified during an input or output operation on a special file. One of these events occurred:

- A device was specified that does not exist, or a request was made beyond the limits of the device.
- The fileset containing the requestor's current working directory or root directory is not mounted. This error can occur after failure and restart of an OSS name server process until the fileset has been repaired and remounted.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation on an input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **chown(2)**, **fstat(2)**, **link(2)**, **mknod(2)**, **open(2)**, **open64(2)**, **pipe(2)**, **utime(2)**.

Miscellaneous Topics: **acl(5)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with this exception:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fstatz -** Provides information about an open file (serializes I/O operations on an open file)

## LIBRARY

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <sys/types.h>    /* optional except for POSIX.1 */
#include <sys/stat.h>
#include <spthread.h>

int spt_fstatz(
        int filedes,
        struct stat *buffer);
```

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**,**dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*buffer* Points to a **stat** structure, into which information is placed about the file. The **stat** structure is described in the **sys/stat.h** header file.

## DESCRIPTION

The **spt_fstatz( )** function is a thread-aware version of the **fstat( )** function.

The **spt_fstatz( )** function obtains information about the open file associated with the *filedes* parameter.

The file information is written to the area specified by the *buffer* parameter, which is a pointer to a **stat** structure. For J06.11 and later J-series RVUs and H06.22 and later H-series RVUs, the **stat** structure uses this definition from the **sys/stat.h** header file:

```
struct stat {
        dev_t     st_dev;
        ino_t     st_ino;
        mode_t    st_mode;
        nlink_t   st_nlink;
        unsigned int      st_acl:1;
        unsigned int      __filler_1:7;
        unsigned int      st_fileprivs:8; /* File privileges */
        uid_t     st_uid;
        gid_t     st_gid;
#if _FILE_OFFSET_BITS != 64 || _TANDEM_ARCH_ == 0
        mode_t    st_basemode; /* Permissions with original group perms */
#endif
        dev_t     st_rdev;
        off_t     st_size;
        time_t    st_atime;
        time_t    st_mtime;
        time_t    st_ctime;
#if _FILE_OFFSET_BITS == 64 && _TANDEM_ARCH_ != 0
        mode_t    st_basemode; /* Permissions with original group perms */
#endif
        int64_t   st_reserved[3];
};
```

For J06.10 and earlier J-series RVUs and H06.21 and earlier H-series RVUs, the **stat** structure uses this definition from the **sys/stat.h** header file:

```
struct  stat {
        dev_t           st_dev;
        ino_t           st_ino;
        mode_t          st_mode;
        nlink_t         st_nlink;
        unsigned int    st_acl:1;
        unsigned int    __filler_1:15;
        uid_t           st_uid;
        gid_t           st_gid;
#if _FILE_OFFSET_BITS != 64 || _TANDEM_ARCH_ == 0
        mode_t          st_basemode; /* Permissions with original group perms */
#endif
        dev_t           st_rdev;
        off_t           st_size;
        time_t          st_atime;
        time_t          st_mtime;
        time_t          st_ctime;
#if _FILE_OFFSET_BITS == 64 && _TANDEM_ARCH_ != 0
        mode_t          st_basemode; /* Permissions with original group perms */
#endif
        int64_t         st_reserved[3];
};
```

The **spt_fstatz( )** function updates any time-related fields associated with the file before writing into the **stat** structure, unless it is a read-only fileset. Time-related fields are not updated for read-only OSS filesets.

The fields in the **stat** structure have these meanings and content:

**st_dev** OSS device identifier for a fileset.

Values for local OSS objects are listed next. Values for local Guardian objects are described in **Use on Guardian Objects**, and values for remote Guardian or OSS objects are described in **Use on Remote Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | ID of device containing directory entry |
| Directory | ID of device containing directory |
| Pipe or FIFO | ID of special fileset for pipes |
| **AF_INET** or **AF_INET6** socket | ID of special fileset for sockets |
| **AF_UNIX** socket | ID of device containing the fileset in which the socket file was created |
| **/dev/null** | ID of device containing directory entry |
| **/dev/tty** | ID of device containing directory entry |

**st_ino** File serial number (inode number). The file serial number and OSS device identifier uniquely identify a regular OSS file within an OSS fileset.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | File serial number (unique) |
| Directory | File serial number (unique) |
| Pipe or FIFO | File serial number (unique) |
| **AF_INET** or **AF_INET6** socket | File serial number (not unique within the HP NonStop node) |
| **AF_UNIX** socket | File serial number of the socket file (unique) |
| **/dev/null** | File serial number (unique) |
| **/dev/tty** | File serial number (unique) |

The **st_ino** value for all node entries in **/E** (including the entry for the logical link from the local node name to the root fileset on the local node) is the value for the root fileset on the corresponding node. If normal conventions are followed, this value is always 0 (zero), so entries in **/E** appear to be nonunique. Values for objects on remote nodes are unique only among the values for objects within the same fileset on that node.

**st_mode** File mode. These bits are ORed into the **st_mode** field:

**S_IFMT** File type. This field can contain one of these values:

**S_IFCHR** Character special file.

**S_IFDIR** Directory.

**S_IFIFO** Pipe or FIFO.

**S_IFREG** Regular file.

**S_IFSOCK** Socket.

For an **AF_INET** or **AF_INET6** socket, the user default permissions are returned for the permission bits. The access flags are set to read and write.

For an **AF_UNIX** socket, the user permissions from the inode for the socket are returned for the permission bits. The access flags are also returned from the inode.

**S_IRWXG** Permissions for the owning group, or if the **st_acl** flag is set, permissions for the the **class** ACL entry.

**S_IRWXO** Other class

**S_IRWXU** Owner class

**S_ISGID** Set group ID on execution

**S_ISUID** Set user ID on execution

**S_ISVTX** Sticky bit; used only for directories (not ORed for files in **/G**, the Guardian file system)

**S_TRUST** Indicates that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers when the memory segment containing the buffers is not shared. This flag applies only to loadfiles for a process, and only a user with appropriate privileges (the super ID) can set it.

**S_TRUSTSHARED**
Indicates that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers regardless of whether the memory segment containing the buffers is shared. This flag applies only to loadfiles for a process, and only a user with appropriate privileges (the super ID) can set it.

Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

**st_nlink** Number of links.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Number of links to the file |
| Directory | Number of links to the directory |
| FIFO | Number of links to the file |
| Pipe | -1 |
| **AF_INET** or **AF_INET6** socket | 0 (zero) |
| **AF_UNIX** socket | Number of links to the socket file |
| **/dev/null** | Number of links to the file |
| **/dev/tty** | Number of links to the file |

**st_acl** If set to 1, indicates that the file has optional access control list (ACL) entries. For compatibility with HP-UX, the member name **st_aclv** is provided as alias for **st_acl**. For more information about ACLs, see the **acl(5)** reference page.

**st_fileprivs** File privileges. For information about file privileges see the **setfilepriv(2)** reference page.

**st_uid** User ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | User ID of the file owner |
| Directory | User ID of the file owner |
| Pipe or FIFO | User ID of the file owner |
| **AF_INET** or **AF_INET6** socket | User ID of the calling process |
| **AF_UNIX** socket | User ID of the creator of the socket file |
| **/dev/null** | User ID of the super ID |
| **/dev/tty** | User ID of the super ID |

**st_gid** Group ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Group ID of the file group |
| Directory | Group ID of the file group |
| Pipe or FIFO | Group ID of the file group |
| **AF_INET** or **AF_INET6** socket | Group ID of the calling process |
| **AF_UNIX** socket | Group ID of the creator of the socket file |
| **/dev/null** | Group ID of the super ID |
| **/dev/tty** | Group ID of the super ID |

**st_basemode** If the **st_acl** flag is set, contains the permissions for the file owner, owning group, and others. If the **st_acl** flag is not set, **st_basemode** is 0 (zero).

**st_rdev** Remote device ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Undefined |
| Directory | Undefined |
| Pipe or FIFO | Undefined |
| **AF_INET** or **AF_INET6** socket | 0 (zero) |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | Undefined |
| **/dev/tty** | ID of the device |

**st_size** File size.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Size of the file in bytes |
| Directory | 4096 |
| Pipe or FIFO | 0 (zero) |
| **AF_INET** or **AF_INET6** socket | 0 (zero) |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | 0 (zero) |
| **/dev/tty** | 0 (zero) |

**st_atime** Access time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last access |
| Directory | Time of the last access |
| Pipe or FIFO | Time of the last access |
| **AF_INET** or **AF_INET6** socket | Value maintained in the socket data structure |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_mtime** Modification time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last data modification |
| Directory | Time of the last modification |
| Pipe or FIFO | Time of the last data modification |
| **AF_INET** or **AF_INET6** socket | Value maintained in the socket data structure |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_ctime** Status change time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last file status change |
| Directory | Time of the last file status change |
| Pipe or FIFO | Time of the last file status change |
| **AF_INET** or **AF_INET6** socket | Value maintained in the socket data structure |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**Use on Guardian Objects**

The **st_dev** and **st_ino** fields of the **stat** structure do not uniquely identify Guardian files (files in **/G**).

The **st_dev** field is unique for **/G**, for each disk volume, and for each Telserv process (or other process of subdevice type 30), because each of these is a separate fileset.

The **S_ISGUARDIANOBJECT** macro can indicate whether an object is a Guardian object when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is a Guardian object and **FALSE** otherwise.

The **st_ino** field is a nonunique encoding of the Guardian filename.

The **st_rdev** field contains a unique minor device number for each entry in **/G/ztnt/**, representing each Telserv process subdevice.

The **st_size** field of an EDIT file (file code 101) is the actual (physical) end of file, not the number of bytes in the file. For directories, **st_size** is set to 4096.

When an OSS function is called for a Guardian EDIT file, the **st_mtime** field is set to the last modification time. The **st_atime** field indicates the last time the file was opened, and the **st_ctime** field is set equal to **st_mtime**. No other time-related fields are updated by OSS function calls.

The **st_ctime** and **st_atime** fields for Guardian regular disk files (except for EDIT files) are updated by OSS function calls, not by Guardian procedure calls.

The time fields for **/G**, **/G/***vol*, and **/G/***vol***/***subvol* always contain the current time.

The mapping between Guardian files and their corresponding file types described in the **st_mode** field is listed next:

| Example in /G | Guardian File Type | st_mode File Type | Permissions |
|---|---|---|---|
| **/G** | N/A | Directory | r-xr-xr-x |
| *vol* | Disk volume | Directory | rwxrwxrwx |
| *vol*/*subvol* | Subvolume | Directory | rwxrwxrwx |
| *vol*/*subvol*/*fileid* | Disk file | Regular file | See following text |
| *vol*/**#123** | Temporary disk file | Regular file | See following text |
| **ztnt** | Subtype 30 process | Directory | --x--x--x |
| **ztnt/#pty0001** | Subtype 30 process with qualifier | Character special | rw-rw-rw- |
| **vol1/zyq00001** | Subvolume | Directory | --------- |

A Guardian file classified as a directory is always owned by the super ID.

Guardian permissions are mapped as follows:

- Guardian network or any user permission is mapped to OSS other permission.
- Guardian community or group user permission is mapped to OSS group permission.
- Guardian user or owner permission is mapped to OSS owner permission.
- Guardian super ID permission is OSS super ID permission.
- Guardian read permission is mapped to OSS read permission.
- Guardian write permission is mapped to OSS write permission.
- Guardian execute permission is mapped to OSS execute permission.
- Guardian purge permission is ignored.

Users are not allowed read access to Guardian processes.

OSS file permissions are divided into three groups (owner, group, and other) of three permission bits each (read, write, and execute). The OSS permission bits do not distinguish between remote and local users as Guardian security does; local and remote users are treated alike.

### Use on Remote Objects

The content of the **st_dev** field of the **stat** structure is unique for each node in **/E** because each node is a separate fileset. Values for directories within **/E** are the same as values for objects on the local HP NonStop node.

The **S_ISEXPANDOBJECT** macro can indicate whether an object in the **/E** directory is on a remote HP NonStop server node when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is on a remote HP NonStop node and **FALSE** otherwise.

## NOTES

This function serializes file operations on an open file. If a thread calls **spt_fstatz( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **fstat( )** to **spt_fstatz( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **fstat( )** to **spt_fstatz( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **spt_fstatz( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not a valid file descriptor.

[EFAULT] The *buffer* parameter points to a location outside of the allocated address space of the process.

[EFSBAD] The program attempted an operation involving a fileset with a corrupted fileset catalog.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOROOT] The program attempted an operation while the root fileset was unavailable.

[ENXIO] An invalid device or address was specified during an input or output operation on a special file. One of these events occurred:

- A device was specified that does not exist, or a request was made beyond the limits of the device.
- The fileset containing the requestor's current working directory or root directory is not mounted. This error can occur after failure and restart of an OSS name server process until the fileset has been repaired and remounted.

- The file size (in bytes) or the file inode number (serial number) cannot be represented correctly in the structure pointed to by the *buffer* parameter.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation on an input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **chown(2)**, **spt_fstatz64(2)**, **link(2)**, **mknod(2)**, **open(2)**, **open64(2)**, **pipe(2)**, **utime(2)**.

Miscellaneous Topics: **acl(5)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with this exception:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- For files other than regular disk files, the **st_size** field of the **stat** structure is set to 0 (zero). For directories, **st_size** is set to 4096.
- The **S_IRWXU**, **S_IRWXG**, **S_IRWXO**, **S_IFMT**, **S_ISVTX**, **S_ISGID**, and **S_ISUID** bits are ORed into the **st_mode** field of the **stat** structure.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EFAULT], [EFSBAD], [EIO], [EISGUARDIAN], [ENETDOWN], [ENOROOT], [ENXIO], and [EWRONGID] can be returned by the **spt_fstatz( )** function.

**NAME**

**spt_fsyncz -** Writes modified data and file attributes to permanent storage (thread-aware version)

**LIBRARY**

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

**SYNOPSIS**

**#include <unistd.h>**
**#include <spthread.h>**

**int spt_fsyncz(**
    **int** *filedes***);**

**PARAMETERS**

*filedes* Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( ),** ,**dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

**DESCRIPTION**

The **spt_fsyncz( )** function is a thread-aware version of the **fsync( )** function.

The **spt_fsyncz( )** function saves all modifications for the file open specified by the *filedes* parameter. On return from the **spt_fsyncz( )** function, all updated data and file attributes have been saved on permanent storage.

**Use on Guardian Objects**

The *filedes* parameter can specify any regular file in **/G** including Guardian EDIT files. Time values are not saved for other file types in **/G**, such as terminal files.

**NOTES**

The **spt_fsyncz( )** function offers an alternative to the **O_SYNC** file status flag. Using **spt_fsyncz( )** calls gives an application control over the performance tradeoffs involved in guaranteeing data integrity. OSS file-system caching can be used for files that are protected only by **spt_fsyncz( )** function calls.

This is a thread-aware function: if this function must wait for an I/O operation to complete on an open file, this function blocks the thread that called it (instead of the entire process), while it waits for the I/O operation to complete.

This function serializes file operations on an open file. If a thread calls **spt_fsyncz( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **fsync( )** to **spt_fsyncz( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **fsync( )** to **spt_fsyncz( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.

- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

Upon successful completion, the **spt_fsyncz( )** function returns the value 0 (zero). Otherwise, it returns the value -1, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **spt_fsyncz( )** function sets **errno** to the value that corresponds to the condition:

[EBADF] The *filedes* parameter is not a valid file descriptor.

[EINTR] The **spt_fsyncz( )** function was interrupted by a signal that was caught.

[EINVAL] The *filedes* parameter, although valid, does not refer to a file on which this operation is possible.

[EIO] An I/O error occurred during a write to the fileset.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENXIO] No such device or address. An invalid device or address was specified during an input or output operation on a special file. One of these events occurred:

- A device was specified that does not exist, or a request was made beyond the limits of the device.
- The fileset containing the requestor's current working directory or root directory is not mounted. This error can occur after failure and restart of an OSS name server process until the fileset has been repaired and remounted.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **open(2)**, **socket(2)**, **stat(2)**, **write(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with this exception:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EISGUARDIAN], [ENETDOWN], [ENXIO], and [EWRONGID] can be returned.

## NAME

**spt_ftruncate64z -** Changes file length (thread-aware version)

## LIBRARY

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <sys/types.h>
#include <spthread.h>

int spt_ftruncate64z(
        int filedes,
        off64_t length);
```

## PARAMETERS

*filedes* Specifies the descriptor of a file that must be open for writing.

*length* Specifies the new length of the file in bytes.

## DESCRIPTION

The **spt_ftruncate64z( )** function is a thread-aware version of the **ftruncate64( )** function.

The **spt_ftruncate64z( )** function changes the length of a file to the size, in bytes, specified by the *length* parameter.

If the new length is less than the previous length, the **spt_ftruncate64z( )** function removes all data beyond *length* bytes from the specified file. All file data between the new EOF and the previous EOF is discarded.

If the new length is greater than the previous length, zeros are added between the previous EOF and the new EOF. If the new length would exceed the file size limit for the calling process, the call to **spt_ftruncate64z( )** fails, and **errno** is set to [EFBIG].

Full blocks are returned to the fileset so that they can be used again, and the file size is changed to the value of the *length* parameter.

The **spt_ftruncate64z( )** function has no effect on First-in, First out (FIFO) special files. This function does not modify the seek pointer of the file. If **spt_ftruncate64z( )** is called for a FIFO file, the call fails, and **errno** is set to [EINVAL].

Upon successful completion, the **spt_ftruncate64z( )** function marks the **st_ctime** and **st_mtime** fields of the file for update. If the file is a regular file, the **spt_ftruncate64z( )** function clears the **S_ISUID** and **S_ISGID** attributes of the file.

## NOTES

The **spt_ftruncate64z( )** function offers an alternative to the **O_SYNC** file status flag. Using **spt_ftruncate64z( )** calls gives an application control over the performance tradeoffs involved in guaranteeing data integrity. OSS file-system caching can be used for files that are protected only by **spt_ftruncate64z( )** function calls.

This function is thread-aware: if this function must wait for an I/O operation to complete on an open file, this function blocks the thread that called it (instead of the entire process), while it waits for the I/O operation to complete.

This function serializes file operations on an open file. If a thread calls **spt_ftruncate64z( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **ftruncate( )** to **spt_ftruncate64z( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** and the **#define _FILE_OFFSET_BITS 64** preprocessor directives before including **spthread.h** or when you use equivalent compiler command options to compile the application.

For C++ applications, an alias to map **ftruncate( )** to **spt_ftruncate64z( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** and the **#define _FILE_OFFSET_BITS 64** preprocessor directives before including **spthread.h** or when you use equivalent compiler command options to compile the application.

For C applications, a macro to map **ftruncate64( )** to **spt_ftruncate64z( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** and the **#define _LARGEFILE64_SOURCE 1** preprocessor directives before including **spthread.h** or when you use equivalent compiler command options to compile the application.

For C++ applications, an alias to map **ftruncate64( )** to **spt_ftruncate64z( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** and the **#define _LARGEFILE64_SOURCE 1** preprocessor directives before including **spthread.h** or when you use equivalent compiler command options to compile the application.

A direct application call to **spt_ftruncatez( )** is automatically mapped to **spt_ftruncate64z( )** when you use the **#define _FILE_OFFSET_BITS 64** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **spt_ftruncate64z( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter does not specify a valid file descriptor open for writing.

[EFBIG] The *length* parameter is greater than the minimum of 2 gigabytes minus 1 byte and the maximum file size established during file open.

[EINTR] The function was interrupted by a signal before any data arrived.

[EINVAL] One of these conditions occurred:

- The file pointed to by the *filedes* parameter is not a regular file.
- The value specified for the *length* parameter was less than 0 (zero).

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to read from its controlling terminal, the process is ignoring or blocking the **SIGTTIN** signal, or the process group is orphaned.
- A physical I/O error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[EROFS] The file resides on a read-only fileset.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

**RELATED INFORMATION**

Functions: **chmod(2)**, **fcntl(2)**, **open(2)**, **open64(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with this exception:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_ftruncatez** - Changes file length (thread-aware version)

## LIBRARY

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <sys/types.h>**
**#include <spthread.h>**

**int spt_ftruncatez(**
    **int** *filedes***,**
    **off_t** *length***);**

## PARAMETERS

*filedes* Specifies the descriptor of a file that must be open for writing.

*length* Specifies the new length of the file in bytes.

## DESCRIPTION

The **spt_ftruncatez( )** is a thread-aware version of the **ftruncate( )** function.

The **spt_ftruncatez( )** function changes the length of a file to the size, in bytes, specified by the *length* parameter.

If the new length is less than the previous length, the **spt_ftruncatez( )** function removes all data beyond *length* bytes from the specified file. All file data between the new EOF and the previous EOF is discarded.

If the new length is greater than the previous length, zeros are added between the previous EOF and the new EOF.

Full blocks are returned to the fileset so that they can be used again, and the file size is changed to the value of the *length* parameter.

The **spt_ftruncatez( )** function has no effect on First-in, First-out (FIFO) special files. This function does not modify the seek pointer of the file. If **spt_ftruncatez( )** is called for a FIFO file, the call fails, and **errno** is set to [EINVAL].

Upon successful completion, the **spt_ftruncatez( )** function marks the **st_ctime** and **st_mtime** fields of the file for update. If the file is a regular file, the **spt_ftruncatez( )** function clears the **S_ISUID** and **S_ISGID** attributes of the file.

## NOTES

The **spt_ftruncatez( )** function offers an alternative to the **O_SYNC** file status flag. Using **spt_ftruncatez( )** calls gives an application control over the performance tradeoffs involved in guaranteeing data integrity. OSS file-system caching can be used for files that are protected only by **spt_ftruncatez( )** function calls.

This function is thread-aware: if this function must wait for an I/O operation to complete on an open file, this function blocks the thread that called it (instead of the entire process), while it waits for the I/O operation to complete.

This function serializes file operations on an open file. If a thread calls **spt_ftruncatez( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **ftruncate( )** to **spt_ftruncatez( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **ftruncate( )** to **spt_ftruncatez( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **spt_ftruncatez( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter does not specify a valid file descriptor open for writing.

[EFBIG] The *length* parameter is greater than the minimum of 2 gigabytes minus 1 byte and the maximum file size established during file open.

[EINTR] The function was interrupted by a signal before any data arrived.

[EINVAL] One of these conditions occurred:

- The file pointed to by the *filedes* parameter is not a regular file.
- The value specified for the *length* parameter was less than 0 (zero).

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to read from its controlling terminal, the process is ignoring or blocking the **SIGTTIN** signal, or the process group is orphaned.
- A physical I/O error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[EROFS] The file resides on a read-only fileset.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **chmod(2)**, **fcntl(2)**, **spt_ftruncatez64(2)**, **open(2)**, **open64(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with this exception:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EISGUARDIAN], [ENETDOWN], and [EROFS] can be returned.

## NAME

**spt_fwrite** - Initiates thread-aware **fwrite( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**ssize_t spt_fwrite(**
  **const void ***pointer**,**
  **size_t** *size***,**
  **size_t** *num_items***,**
  **FILE ***stream***);**

## PARAMETERS

See the **fwrite(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware **fwrite( )** function. The file descriptor underlying the stream must be non-blocking for this function to be thread aware.

The following macro maps **spt_fwrite( )** to **fwrite( )** and has been defined in **spthread.h**:

**#define fwrite(pointer, size, num_items, stream)**
  **spt_fwrite(pointer, size, num_items, stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **fwrite(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying stream becomes invalid (is closed by another thread), 0 is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, 0 is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_fwritex** - Writes to an output stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
[#include <stdio.h>]
#include <spthread.h>

size_t spt_fwritex (
        const void *pointer,
        size_t size,
        size_t num_items,
        FILE *stream
        );
```

## PARAMETERS

*pointer* Points to an array.

*size* Specifies the size of the variable type of the array pointed to by the *pointer* parameter.

*num_items* Specifies the number of items of data.

*stream* Specifies the output stream.

## DESCRIPTION

The **spt_fwritex( )** function is the thread-aware version of the **fwrite( )** function.

The **spt_fwritex( )** function appends *num_items* of data of length *size* from the array pointed to by the *pointer* parameter to the output stream.

The **spt_fwritex( )** function stops writing bytes if an error condition is encountered on the stream, or when the number of items of data specified by the *num_items* parameter have been written. The **spt_fwritex( )** function does not change the contents of the array pointed to by the *pointer* parameter.

## NOTES

The macro to map **fwrite( )** to **spt_fwritex( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **fwrite( )** to **spt_fwritex( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

Upon successful completion, the **spt_fwritex( )** function returns the number of items actually transferred. If the *num_items* parameter is negative or 0 (zero), no characters are transferred, and a value of 0 (zero) is returned. If a write error occurs, the error indicator for the stream is set, and **errno** is set to indicate the error.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), 0 (zero) is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, 0 (zero) is returned with an **errno** value of [EINTR].

## ERRORS

The **spt_fwritex( )** function fails if:

- The *stream* parameter is not open for writing.
- The output file size cannot be increased.
- The *stream* is unbuffered.
- The buffer of *stream* needs to be flushed and the function call causes an underlying **spt_writex( )** or **lseek( )** to be invoked, and this underlying operation fails.

In addition, if any of these conditions occur, the **spt_fwritex( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying the output stream and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying the output stream is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EINTR] The write operation was interrupted by a signal that was caught, and no data was transferred.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned. This error might also be returned under implementation-defined conditions.

[ENOMEM] Insufficient memory storage space is available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

Any error encountered during the underlying call to the **spt_writex( )** function can cause this function to return the corresponding **errno** value reported by the **spt_writex( )** function. If your application program encounters an **errno** value not listed on this reference page, see the **spt_writex(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the cause of that error.

## RELATED INFORMATION

Functions: **fopen(3)**, **fread(3)**, **fwrite(3)**, **getc(3)**, **gets(3)**, **printf(3)**, **putc(3)**, **puts(3)**, **read(2)**, **scanf(3)**, **spt_freadx(2)**, **spt_getcx(2)**, **spt_getsx(2)**, **spt_printfx(2)**, **spt_putcx(2)**, **spt_putsx(2)**, **spt_readx(2)**, **spt_writex(2)**, **write(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_generateTag -** Increments and returns a static long tag

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**long spt_generateTag(void);**

## PARAMETERS

None.

## DESCRIPTION

Increments and returns a static long string appropriate for use as a tag. Note that this long string will eventually wrap, thereby returning tags that may still be in use. For example, if a process calls **spt_generateTag( )** 100 times per second, every second, the wrap will occur on the 248th day.

## RETURN VALUES

This funciton returns a long tag.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_getc -** Initiates thread-aware **getc( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_getc(**
    **FILE ****stream***);**

## PARAMETERS

See the **getc(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **getc( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread aware.

The following macro maps **spt_fgetc( )** to **fgetc( )** and has been defined in **spthread.h**:

**#define getc(stream) spt_getc(stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **getc(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying stream becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_getchar -** Executes thread-aware **getchar( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_getchar(void);**

## PARAMETERS

See the **getchar(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **getchar( )** function. The file descriptor underlying standard input must be nonblocking for this function to be thread aware.

The following macro maps **spt_getchar( )** to **getchar( )** and has been defined in **spthread.h**:

**#define getchar() spt_getchar()**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **getchar(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying standard input becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill(2** function and is not blocked, ignored, or handled, EOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_getcharx -** Gets a character from the standard input stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

**int spt_getcharx (void);**

## PARAMETERS

None.

## DESCRIPTION

The **spt_getcharx( )** function is the thread-aware version of the **getchar( )** function.

The **spt_getcharx( )** function returns the next byte from the standard input stream and moves the file pointer, if defined, ahead one byte.

## NOTES

The macro to map **getchar( )** to **spt_getcharx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **getchar( )** to **spt_getcharx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

The **spt_getcharx( )** function might be a macro (depending on the compile-time definitions used in the source). Consequently, you cannot use this interface where a function is necessary; for example, a subroutine pointer cannot point to it. When a function is necessary, use the **spt_fgetcx( )** function instead.

## RETURN VALUES

This function and macro returns a character if successful. It returns the integer constant **EOF** at the end of the file or upon an error. The function sets **errno** when an error is encountered.

If the file descriptor underlying **stdin** becomes invalid (is closed by another thread), **EOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **EOF** is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_getcharx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the underlying input stream and the process would be delayed by the read operation.

[EBADF] The file descriptor underlying the input stream is not a valid file descriptor or is not open for reading.

[EINTR] The read operation was interrupted by a signal that was caught, and no data was transferred.

[ENXIO] A request was made on a nonexistent device, or the request was outside the capabilities of the device.

[EIO] The call is attempting to read from the process's controlling terminal and either the process is ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.

[ENOMEM] Insufficient memory is available for the operation.

Any error encountered during the underlying call to the **spt_readx( )** function can cause this function to return the corresponding **errno** value reported by the **spt_readx( )** function. If your application program encounters an **errno** value not listed on this reference page, see the **spt_readx(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the cause of that error.

**RELATED INFORMATION**

Functions: **fgetc(3)**, **getc(3)**, **getchar(3)**, **gets(3)**, **getwc(3)**, **putc(3)**, **read(2)**, **spt_fgetcx(2)**, **spt_getcx(2)**, **spt_getsx(2)**, **spt_getwcx(2)**, **spt_putcx(2)**, **spt_readx(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_getcx -** Gets a character from a specified input stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

**int spt_getcx (**
    **FILE** **stream*
    **);**

## PARAMETERS

*stream* Points to the **FILE** structure of an open file.

## DESCRIPTION

The **spt_getcx( )** function is the thread-aware version of the **getc( )** function.

The **spt_getcx( )** function returns the next byte from the input specified by the *stream* parameter and moves the file pointer, if defined, ahead one byte in *stream*.

## NOTES

The macro to map **getc( )** to **spt_getcx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **getc( )** to **spt_getcx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

The **spt_getcx( )** function might be a macro (depending on the compile-time definitions used in the source). Consequently, you cannot use this interface where a function is necessary; for example, a subroutine pointer cannot point to it. In addition, **spt_getcx( )** does not work correctly with a *stream* parameter that has side effects. In particular, the following does not work:

```
spt_getcx(*f++)
```

When a function is necessary, use the **spt_fgetcx( )** function instead.

## RETURN VALUES

This function and macro:

- Returns a character if successful.
- Returns the integer constant **EOF** at the end of the file or upon an error.
- Sets **errno** when an error is encountered.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), **EOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **EOF** is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_getcx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the underlying input stream and the process would be delayed by the read operation.

[EBADF] The file descriptor underlying the input stream is not a valid file descriptor or is not open for reading.

[EINTR] The read operation was interrupted by a signal that was caught, and no data was transferred.

[ENXIO] A request was made on a nonexistent device, or the request was outside the capabilities of the device.

[EIO] The call is attempting to read from the process's controlling terminal and either the process is ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.

[ENOMEM] Insufficient memory is available for the operation.

Any error encountered during the underlying call to the **spt_readx( )** function can cause this function to return the corresponding **errno** value reported by the **spt_readx( )** function. If your application program encounters an **errno** value not listed on this reference page, see the **spt_readx(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the cause of that error.

## RELATED INFORMATION

Functions: **fgetc(3)**, **getchar(3)**, **getc(3)**, **gets(3)**, **getwc(3)**, **putc(3)**, **read(2)**, **spt_fgetcx(2)**, **spt_getcharx(2)**, **spt_getsx(2)**, **spt_getwcx(2)**, **spt_putcx(2)**, **spt_readx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_gets -** Initiates thread-aware **gets( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_gets(**
    **FILE** ******stream***);**

## PARAMETERS

See the **gets(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **gets(3)** function. The file descriptor underlying standard input must be nonblocking for this function to be thread aware.

The following macro maps **spt_gets( )** to **gets( )** and has been defined in **spthread.h**:

**#define gets(string) spt_gets(string)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **gets(3)** reference page. The following information also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying standard input becomes invalid (is closed by another thread), NULL is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, NULL is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_getsx -** Gets a string from the standard input stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

```
[#include <stdio.h>]
#include <spthread.h>

char *spt_getsx (
        char *string
        );
```

## PARAMETERS

*string* Points to a string to receive bytes.

## DESCRIPTION

The **spt_getsx( )** function is the thread-aware version of the **gets( )** function.

The **spt_getsx( )** function reads bytes from the standard input stream, **stdin**, into the array pointed to by the *string* parameter. Data is read until a newline character is read or an end-of-file condition is encountered. If reading is stopped due to a newline character, the newline character is discarded and the string is terminated with a NULL character.

## NOTES

The macro to map **gets( )** to **spt_getsx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **gets( )** to **spt_getsx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

The **spt_getsx( )** function does not check the input for a maximum size. Consequently, if more bytes are entered than will fit in the space allocated for the *string* parameter, **spt_getsx( )** will write beyond the end of the allocated space, producing indeterminate results. To avoid this condition, you should use **spt_fgetsx( )** instead of **spt_getsx( )**.

## RETURN VALUES

If the end of the file is encountered and no characters have been read, no characters are transferred to *string* and a null pointer is returned. If a read error occurs, a null pointer is returned. Otherwise, *string* is returned.

If the file descriptor underlying **stdin** becomes invalid (is closed by another thread), **NULL** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **NULL** is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_getsx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the underlying input stream and the process would be delayed by the read operation.

[EBADF] The file descriptor underlying the input stream is not a valid file descriptor or is not open for reading.

[EINTR] The read operation was interrupted by a signal that was caught, and no data was transferred.

[ENXIO] A request was made on a nonexistent device, or the request was outside the capabilities of the device.

[EIO] The call is attempting to read from the process's controlling terminal and either the process is ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.

[ENOMEM] Insufficient memory is available for the operation.

Any error encountered during the underlying call to the **spt_readx( )** function can cause this function to return the corresponding **errno** value reported by the **spt_readx( )** function. If your application program encounters an **errno** value not listed on this reference page, see the **spt_readx(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the cause of that error.

## RELATED INFORMATION

Functions: **clearerr(3)**, **feof(3)**, **ferror(3)**, **fgets(3)**, **fileno(3)**, **fopen(3)**, **fread(3)**, **getc(3)**, **gets(3)**, **getwc(3)**, **puts(3)**, **scanf(3)**, **spt_freadx(2)**, **spt_getcx(2)**, **spt_gets(2)**, **spt_getwcx(2)**, **spt_putsx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_getTMFConcurrentTransactions -** Gets the number of concurrent TMF transactions being used

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_getTMFConcurrentTransactions ( void );**

## PARAMETERS

None.

## DESCRIPTION

This function gets the number of concurrent TMF transactions being used.

## RETURN VALUES

Upon successful completion, this function returns as an integer value the number of transactions being used.

## RELATED INFORMATION

Functions: **spt_setTMFConcurrentTransactions(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_getw -** Initiates thread-aware **getw( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_getw(**
    **FILE** ***stream*);**

## PARAMETERS

See the **getw(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **getw( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread aware.

The following macro maps **spt_getw( )** to **getw( )** and has been defined in **spthread.h**:

**#define getw(stream) spt_getw(stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **getw(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying the stream becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_getwc -** Initiates thread-aware **getwc( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**wint_t spt_getwc(**
    **FILE ***stream***);**

## PARAMETERS

See the **getwc(3)** reference page either online or in the *Open System Services Library Calls Refrence Manual*.

## DESCRIPTION

This is a thread-aware version of the **getwc( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread aware.

The following macro maps **spt_getwc( )** to etwc( ) and has been defined in **spthread.h**:

**#define getwc(stream) spt_getwc(stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **getwc(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying stream becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_getwchar -** Initiates thread-aware **getwchar( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**wint_t spt_getwchar(void);**

## PARAMETERS

See the **getwchar(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **getwchar(3)** function. The file descriptor underlying standard input must be nonblocking for this function to be thread aware.

The following macro maps **spt_getwchar( )** to **getwchar( )** and has been defined in **spthread.h**:

**#define getwchar() spt_getwchar()**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **getwchar(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying standard input becomes invalid (is closed by another thread), WEOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, WEOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_getwcharx -** Gets a wide character from the standard input stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <wchar.h>**]
**#include <spthread.h>**

**wint_t spt_getwcharx (void);**

## PARAMETERS

None.

## DESCRIPTION

The **spt_getwcharx( )** function is the thread-aware version of the **getwchar( )** function.

The **spt_getwcharx( )** function gets the next wide character from the standard input stream. It is equivalent to **spt_getwcx(***stdin***)**.

## NOTES

The macro to map **getwchar( )** to **spt_getwcharx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **getwchar( )** to **spt_getwcharx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

This function returns the wide character read or the constant **WEOF** (wide-character end-of-file) at the end of the file or upon an error.

If the file descriptor underlying **stdin** becomes invalid (is closed by another thread), **WEOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **WEOF** is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_getwcharx( )** function sets **errno** to the corresponding value:

[EBADF] The file descriptor underlying **stdin** is no longer valid.

[EINTR] A signal was received that is not blocked, ignored or handled.

## RELATED INFORMATION

Functions: **fgetwc(3)**, **fopen(3)**, **fread(3)**, **getc(3)**, **gets(3)**, **getwc(3)**, **getwchar(3)**, **putwc(3)**, **scanf(3)**, **spt_fgetwcx(2)**, **spt_freadx(2)**, **spt_fgetcx(2)**, **spt_getsx(2)**, **spt_getwcx(2)**, **spt_putwcx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_getwcx -** Gets a wide character from a specified input stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <wchar.h>**]
**#include <spthread.h>**

**wint_t spt_getwcx (**
    **FILE ****stream*
    **);**

## PARAMETERS

*stream* Specifies the input data.

## DESCRIPTION

The **spt_getwcx( )** function is the thread-aware version of the **getwc( )** function.

The **spt_getwcx( )** function gets the next wide character from the input stream specified by the *stream* parameter.

## NOTES

The macro to map **getwc( )** to **spt_getwcx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **getwc( )** to **spt_getwcx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

This function returns the wide character read or the constant **WEOF** (wide-character end-of-file) at the end of the file or upon an error.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), **WEOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **WEOF** is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_getwcx( )** function sets **errno** to the corresponding value:

[EBADF] The file descriptor underlying *stream* is no longer valid.

[EINTR] A signal was received that is not blocked, ignored, or handled.

## RELATED INFORMATION

Functions: **fgetwc(3)**, **fopen(3)**, **fread(3)**, **getc(3)**, **gets(3)**, **getwc(3)**, **getwchar(3)**, **putwc(3)**, **scanf(3)**, **spt_getcx(2)**, **spt_getsx(2)**, **spt_getwcharx(2)**, **spt_putwcx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_getwx -** Gets a word from an input stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

```
[#include <stdio.h>]
#include <spthread.h>

int spt_getwx (
        FILE *stream
        );
```

## PARAMETERS

*stream* Points to the file structure of an open file.

## DESCRIPTION

The **spt_getwx( )** function is the thread-aware version of the **getw( )** function.

The **spt_getwx( )** function returns the next word (**int**) from the input specified by the *stream* parameter and increments the associated file pointer, if defined, to point to the next **int**.

The **spt_getw( )** function returns the constant **EOF** at the end of the file or when an error occurs. Since **EOF** is a valid integer value, you can use the **feof( )** and **ferror( )** functions to check the success of **spt_getwx( )**. The **spt_getwx( )** function assumes no special alignment in the file.

## NOTES

The macro to map **getw( )** to **spt_getwx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **getw( )** to **spt_getwx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

Because of possible differences in **int** length and byte ordering from one machine architecture to another, files written using the **spt_putwx( )** function are machine dependent and might not be readable using **getw( )** on a different type of processor.

## RETURN VALUES

The **spt_getwx( )** function returns the integer constant **EOF** at the end of the file or upon an error.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), **EOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **EOF** is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_getwx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the underlying *stream* and the process would be delayed by the read operation.

[EBADF] The file descriptor underlying the *stream* is not a valid file descriptor or is not open for reading.

[EINTR] The read operation was interrupted by a signal that was caught, and no data was transferred.

[ENXIO] A request was made on a nonexistent device, or the request was outside the capabilities of the device.

[EIO] The call is attempting to read from the process's controlling terminal and either the process is ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.

[ENOMEM] Insufficient memory is available for the operation.

**RELATED INFORMATION**

Functions: **fgetc(3)**, **getc(3)**, **getchar(3)**, **gets(3)**, **getw(3)**, **getwc(3)**, **putc(3)**, **spt_fgetcx(2)**, **spt_getcx(2)**, **spt_getcharx(2)**, **spt_getsx(2)**, **spt_getwcx(2)**, **spt_putcx(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_INITRECEIVE -** Registers $RECEIVE filename

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**long spt_INITRECEIVE (**
  **const short** *filenum***,**
  **const short** *receive_depth***);**

## PARAMETERS

*filenum* Specifies Guardian file number whose IO has completed

*receive_depth* Specifies the maximum number of incoming messages as specified in the *filenum* value is **FILE_OPEN( )** call

## DESCRIPTION

This function registers *filenum* as being managed by the $RECEIVE callback.

## RETURN VALUES

This function returns Guardian error numbers, which include:

0 $RECEIVE was successfully registered.

29 $RECEIVE was already registered prior to this call.

29 **FILE_COMPLETE_SET_( )** addition of $RECEIVE returned nonzero.

29 Value for *filenum* not 0.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_INITRECEIVEL -** Registers $RECEIVE filename (larger message version)

## LIBRARY

H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**long spt_INITRECEIVEL (**
**const short** *filenum***,**
**const short** *receive_depth***);**

## PARAMETERS

*filenum* Specifies Guardian file number whose IO has completed

*receive_depth* Specifies the maximum number of incoming messages as specified in the *filenum* value is **FILE_OPEN( )** call

## DESCRIPTION

This function is the same as the **spt_INITRECEIVE( )** function, except:

- This function can handle the longer message lengths allowed by the **SPT_SERVERCLASS_SENDL_( )** function.
- The Guardian file-system error 4184 (EVERSION) can be returned.

See the **spt_INITRECEIVE( )** reference page.

## NOTES

This function is supported on systems running J06.07 and later J-series RVUs and H06.18 and later H-series RVUs, and must be used instead of the **spt_INITRECEIVE( )** function when the messages are larger than 32 kilobytes. This function also can be used for shorter messages.

## RETURN VALUES

See the **spt_INITRECEIVE( )** reference page.

In addition, this function can return this Guardian file-system error:

4184 (EVERSION)
The function was called from a system that is running a J-series RVU earlier than J06.07 or an H-series RVU earlier than H06.18.

## RELATED INFORMATION

Functions: **spt_INITRECEIVE(2)**, **SPT_SERVERCLASS_SENDL_(3)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification.

## NAME

**spt_interrupt -** Interrupts all threads awaiting input or output

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**spt_error_t spt_interrupt(**
    **const short** *filenum***,**
    **const spt_error_t** *errorSPT***);**

## PARAMETERS

*filenum* Specifies the Guardian file number for thev file whose awaiting I/O is to be interrupted.

*errorSPT* Specifies SPT error returned to waiting file.

## DESCRIPTION

Interrupts all threads awaiting IO on file number. Note the I/O is not cancelled by this function. Interrupted threads will return from the **spt_awaitio( )** function with a return value of *error_SPT*. Additionally, the *error* parameter passed to the **spt_awaitio( )** function will be set as shown in the **PARAMETERS** section.

## RETURN VALUES

**SPT_SUCCESS**
The file number awaiting I/O (if any) was interrupted.

**SPT_ERROR** Either the value specified for *error_SPT* is invalid or the value for *filenum* is less than 0 (zero) or is not registered.

## ERRORS

-1 **- SPT_ERROR**

40 **- SPT_TIMEOUT**

[EINTR] **- SPT_INTERRUPTED**

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_interruptTag -** Interrupts thread awaiting tagged I/O

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**spt_error_t spt_interruptTag(**
**const short** *filenum***,**
**const long** *tag***,**
**const spt_error_t** *error_SPT***);**

## PARAMETERS

*filenum* Specifies the Guardian file number for the file whose awaiting I/O is to be interrupted

*tag* Specifies tag whose awaiting I/O is to be interrupted

*error_SPT* Specifies SPT error returned to awaiting IO

## DESCRIPTION

Interrupts the thread awaiting the tagged I/O on file number. Note that the I/O is not cancelled by this function. Interrupted threads will return from the **spt_awaitio( )** function with a return value of *error_SPT*. Additionally, the *error* parameter passed to **spt_awaitio( )** will be set as shown in the **ERRORS** section.

## RETURN VALUES

**SPT_SUCCESS**
Awaiting IO was interrupted.

**SPT_ERROR** One of the following conditions exists:

- The value of *filenum* was less than 0 (zero), or no awaiting I/O was found
- The value of *filenum* is not registered
- The value for *error_SPT* is invalid

## ERRORS

-1 **SPT_ERROR**

40 **SPT_TIMEDOUT**

EINTR **SPT_INTERRUPTED**

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_LOCKFILE -** Excludes other users from accessing a Guardian disk file

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_LOCKFILE (**
**short** *filenum***,**
[ **long** *tag* ]
**);**

## PARAMETERS

*filenum* specifies the file number of a Guardian disk file open instance that identifies the file to be locked

*tag* is for nowait input/output (I/O) only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **SPT_LOCKFILE( )** function is the thread-aware version of the Guardian LOCKFILE procedure.

The **SPT_LOCKFILE( )** function is used to exclude other users from accessing a file (and any records within that file). The user is defined either as the opener of the file (identified by filenum) if the file is not audited or as the transaction (identified by the TRANSID) if the file is audited. If the file is currently unlocked or is locked by the current user when **SPT_LOCKFILE( )** is called, the file (and all its records) becomes locked, and the caller continues executing. If the file is already locked by another user, the behavior of the system is specified by the locking mode. Two locking modes are available:

Default The process requesting the lock is suspended. See the **Considerations** subsection of this reference page.

Alternate The lock request is rejected with Guardian file-system error 73. When the alternate locking mode is in effect, the process requesting the lock is not suspended. See the **Considerations** subsection of this reference page.

For programming information about the LOCKFILE procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

### Considerations

Record locking versus file locking

A call to **SPT_LOCKFILE( )** is not equivalent to locking all records in a file; that is, locking all records still allows insertion of new records, but file locking does not. File locks and record locks are queued in the order in which they are issued.

Nowait and **SPT_LOCKFILE( )**

If the **SPT_LOCKFILE( )** function is used to initiate an operation with a file opened for nowait I/O, it must complete with a corresponding call to the Guardian AWAITIO procedure.

Locking modes

Default mode: If the file is already locked by another user when **SPT_LOCKFILE( )** is called, the process requesting the lock is suspended and queued in a locking queue behind other users trying to access the file. When the file becomes unlocked, the user at the head of the locking queue is granted access to the file. If the user at the head of the locking queue is requesting a lock, the user is granted the lock and resumes execution. If the user at the head of the locking queue is requesting a read, the read operation continues to completion.

Alternate mode: If the file is already locked by another user when the call to **SPT_LOCKFILE( )** is made, the lock request is rejected, and the call to **SPT_LOCKFILE( )** completes immediately with Guardian file-system error 73 (`file is locked`). The alternate locking mode is specified by calling the **SPT_SETMODE( )** procedure and specifying function 4.

Locks and open files (applies to nonaudited files only)

Locks are granted on a file open (that is, on a file number) basis. Therefore, if a process has multiple opens of the same file, a lock of one file number excludes access to the file through other file numbers.

Attempting to read a locked file in default locking mode

If the default locking mode is in effect when a call to **SPT_READX( )** or **SPT_READUPDATEX( )** is made for a file that is locked by another user, the caller of **SPT_READX( )** or **SPT_READUPDATEX( )** is suspended and queued in the locking queue behind other users attempting to access the file.

For nonaudited files, a deadlock condition (a permanent suspension of your application) occurs if **SPT_READX( )** or **SPT_READUPDATEX( )** is called by the process that has a record locked with a file number other than that supplied in the **SPT_READX( )** or **SPT_READUPDATEX( )** call. For an explanation of multiple opens by the same process, see the **SPT_FILE_OPEN_(2)** reference page either online or in the *Open System Services System Calls Reference Manual*.

Accessing a locked file

If the file is locked by a user other than the caller at the time of the call, the call is rejected with Guardian file-system error 73 (`file is locked`) when:

**SPT_READX( )** or **SPT_READUPDATEX( )** is called, and the alternate locking mode is in effect.

**SPT_WRITEX( )**, WRITEUPDATE, or **SPT_CONTROL( )** is called.

A count of the locks in effect is not maintained. Multiple locks can be unlocked with one call to **SPT_UNLOCKFILE( )**.

### Use on OSS Objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **SPT_LOCKFILE( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_LOCKREC -** Excludes other users from accessing a record in a Guardian disk file

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_LOCKREC (**
    **short** *filenum***,**
    [ **long** *tag* ]
    **);**

## PARAMETERS

*filenum*
: specifies the file number of a Guardian disk file open instance that identifies the file containing the record to be locked

*tag*
: is for nowait input/output (I/O) only. The *tag* value you define uniquely identifies the operation associated with this call.

  This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **SPT_LOCKREC( )** function is the thread-aware version of the Guardian LOCKREC procedure.

The LOCKREC procedure excludes other users from accessing a record at the current position. The user is defined either as the opener of the file (identified by *filenum*) if the file is not audited or as the transaction (identified by the TRANSID) if the file is audited.

For key-sequenced, relative, and entry-sequenced files, the current position is the record with a key value that matches exactly the current key value. For unstructured files, the current position is the relative byte address (RBA) identified by the current-record pointer. If the record is unlocked when **SPT_LOCKREC( )** is called, the record becomes locked, and the caller continues executing.

You cannot use **SPT_LOCKREC( )** with queue files.

If the file is already locked by another user, the behavior of the system is specified by the locking mode. Two locking modes are available:

Default
: The process requesting the lock is suspended. See the **Considerations** subsection of this reference page.

Alternate
: The lock request is rejected with Guardian file-system error 73. When the alternate locking mode is in effect, the process requesting the lock is not suspended. See the **Considerations** subsection of this reference page.

For programming information about the LOCKREC procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

## Considerations

Record locking versus file locking

A call to **SPT_LOCKFILE()** is not equivalent to locking all records in a file; that is, locking all records still allows insertion of new records, but file locking does not. File locks and record locks are queued in the order in which they are issued.

Nowait and **SPT_LOCKREC()**

If the **SPT_LOCKREC()** function is used to initiate an operation with a file opened for nowait I/O, it must complete with a corresponding call to the Guardian AWAITIO procedure.

Default locking mode

If the record is already locked by another user when **SPT_LOCKREC()** is called, the process requesting the lock is suspended and queued in a locking queue behind other users also requesting to lock or read the record.

When the record becomes unlocked, the user at the head of the locking queue is granted access to the record. If the user at the head of the locking queue is requesting a lock, it is granted the lock and resumes execution. If the user at the head of the locking queue is requesting a read operation, the read operation continues to completion.

Alternate locking mode

If the record is already locked by another user when **SPT_LOCKREC()** is called, the lock request is rejected, and the call to **SPT_LOCKREC()** completes immediately with Guardian file-system error 73 (`record is locked`). The alternate locking mode is specified by calling the **SPT_SETMODE()** procedure and specifying function 4.

Attempting to read a locked record in default locking mode

If the default locking mode is in effect when **SPT_READX()** or **SPT_READUPDATEX()** is called for a record that is locked by another user, the caller to **SPT_READX()** or **SPT_READUPDATEX()** is suspended and queued in the locking queue behind other users attempting to lock or read the record. (Another user means another open *filenum* if the file is not audited, or another TRANSID if the file is audited.)

For nonaudited files, a deadlock condition (a permanent suspension of your application) occurs if **SPT_READX()** or **SPT_READUPDATEX()** is called by the process that has a record locked with a file number other than that supplied in the **SPT_READX()** or **SPT_READUPDATEX()** call. For an explanation of multiple opens by the same process, see the **SPT_FILE_OPEN_(2)** reference page either online or in the *Open System Services System Calls Reference Manual*.

Selecting the locking mode with **SPT_SETMODE()**

The locking mode is specified by the calling SETMODE procedure with function 4.

A count of the locks in effect is not maintained. Multiple locks can be unlocked with one call to **SPT_UNLOCKFILE()**.

Structured files

Calling LOCKREC after positioning on a nonunique key

If the call to **SPT_LOCKREC( )** immediately follows a call to KEYPOSITION where a nonunique alternate key is specified, the call to **SPT_LOCKREC( )** fails. A subsequent call to the Guardian FILE_GETINFO_ or FILEINFO procedure shows that a Guardian file-system error 46 (`invalid key`) occurred. However, if an intermediate call to **SPT_READX( )** is performed, the call to **SPT_LOCKREC( )** is permitted because a unique record is identified.

Current-state indicators after **SPT_LOCKREC( )**

After a successful call to **SPT_LOCKREC( )**, current-state indicators are unchanged.

Unstructured files

Locking the relative byte address (RBA) in an unstructured file

Record positions in an unstructured file are represented by an RBA, and the RBA can be locked with **SPT_LOCKREC( )**. To lock a position in an unstructured file, first call the Guardian POSITION procedure with the desired RBA, and then call **SPT_LOCKREC( )**. This locks the RBA; any other process attempting to access the file with exactly the same RBA encounters a `record is locked` condition. You can access that RBA by positioning to RBA-2. Depending on the process's locking mode, the call either fails with Guardian file-system error 73 (`record is locked`) or is placed in the locking queue.

Record pointers after a call to **SPT_LOCKREC( )**

After a call to **SPT_LOCKREC( )**, the current-record, next-record, and end-of-file pointers remain unchanged.

Ways to avoid or resolve deadlocks

One way to avoid deadlock is to call function 4 of the **SPT_SETMODE( )** procedure to establish one of the alternate locking modes. A common method of avoiding deadlock situations is to lock records in some predetermined order. Deadlocks can be resolved if you lock records using a nowait open and call the Guardian AWAITIO procedure with a timeout specified.

### Use on OSS Objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **SPT_LOCKREC( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_lseek64z -** Sets file offset for read or write operation (serializes I/O operations on an open file)

## LIBRARY

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <sys/types.h>** /* optional except for POSIX.1 */
**#include <unistd.h>**
**#include <spthread.h>**

**off64_t spt_lseek64z(**
 **int** *filedes***,**
 **off64_t** *offset***,**
 **int** *whence***);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*offset* Specifies a value, in bytes, that is used with the *whence* parameter to set the file pointer. A negative value causes seeking in the reverse direction.

*whence* Specifies how to interpret the *offset* parameter in setting the file pointer associated with the *filedes* parameter. Values for the *whence* parameter are:

**SEEK_CUR** Sets the file pointer to its current location plus the value of the *offset* parameter.

**SEEK_END** Sets the file pointer to the size of the file plus the value of the *offset* parameter.

**SEEK_SET** Sets the file pointer to the value of the *offset* parameter.

## DESCRIPTION

The **spt_lseek64z( )** is a thread-aware version of the **lseek64( )** function.

The **spt_lseek64z( )** function sets the file offset for the open file specified by the *filedes* parameter. The *whence* parameter determines how the offset is to be interpreted.

The **spt_lseek64z( )** function allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reading of data in the gap returns bytes with the value 0 (zero) until data is actually written into the gap.

The **spt_lseek64z( )** function does not, by itself, extend the size of the file.

## NOTES

The **spt_lseekz( )** function offers an alternative to the **O_SYNC** file status flag. Using **spt_lseekz( )** calls gives an application control over the performance tradeoffs involved in guaranteeing data integrity. OSS file-system caching can be used for files that are protected only by **spt_lseekz( )** function calls.

This function serializes file operations on an open file. If a thread calls **spt_lseek64z( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **lseek( )** to **spt_lseek64z( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** and the **#define _FILE_OFFSET_BITS 64** preprocessor directives before including **spthread.h** or when you use equivalent compiler command options to compile the application.

For C++ applications, an alias to map **lseek( )** to **spt_lseek64z( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** and the **#define _FILE_OFFSET_BITS 64** preprocessor directives before including **spthread.h** or when you use equivalent compiler command options to compile the application.

For C applications, a macro to map **lseek64( )** to **spt_lseek64z( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** and the **#define _LARGEFILE64_SOURCE 1** preprocessor directives before including **spthread.h** or when you use equivalent compiler command options to compile the application.

For C++ applications, an alias to map **lseek64( )** to **spt_lseek64z( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** and the **#define _LARGEFILE64_SOURCE 1** preprocessor directives before including **spthread.h** or when you use equivalent compiler command options to compile the application.

A direct application call to **spt_lseekz( )** is automatically mapped to **spt_lseek64z( )** when you use the **#define _LARGEFILE64_SOURCE 1** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

### RETURN VALUES

Upon successful completion, the resulting pointer location, measured in bytes from the beginning of the file, is returned. For First-in, First-out (FIFO) files, pipes, and character special files, the value 0 (zero) is returned. For character special files, **errno** is not set.

If the **spt_lseek64z( )** function fails, the file offset remains unchanged, the value -1 cast to the type **off_t** is returned, and **errno** is set to indicate the error.

### ERRORS

If any of these conditions occurs, the file offset remains unchanged, and the **spt_lseek64z( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not an open file descriptor.

[EINVAL] One of these conditions exists:

- The *whence* parameter is an invalid value, or the resulting file offset would be an invalid value (that is, a value less than 0 [zero]).
- The *filedes* parameter refers to a file (other than a pipe, FIFO, or directory) on which seeking cannot be performed.

[EISDIR] The *filedes* parameter refers to an OSS directory.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[EOVERFLOW]
The application attempted to set the file offset beyond the maximum file offset supported for the file.

[ESPIPE] The *filedes* parameter refers to a pipe, FIFO, or socket.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and the backup process took over.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Functions: **fcntl(2)**, **fseek(3)**, **open(2)**, **open64(2)**, **read(2)**, **write(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with this exception:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_lseekz** - Sets file offset for read or write operation (serializes I/O operations on an open file))

## LIBRARY

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <unistd.h>
#include <spthread.h>

off_t spt_lseekz(
        int filedes,
        off_t offset,
        int whence);
```

## PARAMETERS

*filedes*
: Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*offset*
: Specifies a value, in bytes, that is used with the *whence* parameter to set the file pointer. A negative value causes seeking in the reverse direction.

*whence*
: Specifies how to interpret the *offset* parameter in setting the file pointer associated with the *filedes* parameter. Values for the *whence* parameter are:

**SEEK_CUR** Sets the file pointer to its current location plus the value of the *offset* parameter.

**SEEK_END** Sets the file pointer to the size of the file plus the value of the *offset* parameter.

**SEEK_SET** Sets the file pointer to the value of the *offset* parameter.

## DESCRIPTION

The **spt_lseekz( )** is a thread-aware version of the **lseek( )** function.

The **spt_lseekz( )** function sets the file offset for the open file specified by the *filedes* parameter. The *whence* parameter determines how the offset is to be interpreted.

The **spt_lseekz( )** function allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reading of data in the gap returns bytes with the value 0 (zero) until data is actually written into the gap.

The **spt_lseekz( )** function does not, by itself, extend the size of the file.

## NOTES

The **spt_lseekz( )** function offers an alternative to the **O_SYNC** file status flag. Using **spt_lseekz( )** calls gives an application control over the performance tradeoffs involved in guaranteeing data integrity. OSS file-system caching can be used for files that are protected only by **spt_lseekz( )** function calls.

This function serializes file operations on an open file. If a thread calls **spt_lseekz( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **lseek( )** to **spt_lseekz( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **lseek( )** to **spt_lseekz( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

Upon successful completion, the resulting pointer location, measured in bytes from the beginning of the file, is returned. For First-in, First-out (FIFO) files, pipes, and character special files, the value 0 (zero) is returned. For character special files, **errno** is not set.

If the **spt_lseekz( )** function fails, the file offset remains unchanged, the value -1 cast to the type **off_t** is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the file offset remains unchanged, and the **spt_lseekz( )** function sets **errno** to the corresponding value:

[EBADF] The *filedes* parameter is not an open file descriptor.

[EINVAL] One of these conditions exists:

- The *whence* parameter is an invalid value, or the resulting file offset would be an invalid value (that is, a value less than 0 [zero]).
- The *filedes* parameter refers to a file (other than a pipe, FIFO, or directory) on which seeking cannot be performed.

[EISDIR] The *filedes* parameter refers to an OSS directory.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[EOVERFLOW]
The application was compiled in a regular compilation environment or was compiled using the **#define _LARGEFILE64_SOURCE 1** feature test macro (or an equivalent compiler command option), and the application attempted to set the pointer location at a position between 2 gigabytes minus 1 byte and the maximum file offset established when the file was opened.

[ESPIPE] The *filedes* parameter refers to a pipe, FIFO, or socket.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and the backup process took over.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Functions: **fcntl(2)**, **fseek(3)**, **spt_lseekz64(2)**, **open(2)**, **open64(2)**, **read(2)**, **write(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with this exception:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- If the **spt_lseekz( )** function is called for a pipe or FIFO, the **errno** value [ESPIPE] is returned.
- If the **spt_lseekz( )** function is called for a character special file, no **errno** value is returned.
- If the **spt_lseekz( )** function is called for any other device on which seeking cannot be performed, the operation fails, and **errno** is set to [EINVAL].

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EINVAL], [EISDIR], [EISGUARDIAN], and [EWRONGID] can be returned.

## NAME

**spt_OSSFileIOHandler_p -** Executes callback type required by the **spt_regOSSFileIOHandler(** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**typedef void (**
    ***spt_OSSFileIOHandler_p)(const int** *filedes***,**
    **const int** *read***,**
    **const int** *write***,**
    **const int** *error***);**

## PARAMETERS

*filedes* Specifies OSS file descriptor of interest

*read* Specifies file descriptor is read ready

*write* Specifies file descriptor is write ready

*error* Specifies file descriptor has an exception pending

## DESCRIPTION

This function executes the callback type required by the **spt_regOSSFileIOHandler( )** function. This callback is executed in the context of the last running thread (on the stack of the last running thread).

## RETURN VALUES

None.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_pause -** Suspends a thread until a signal is received.

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_pause(void)**

## DESCRIPTION

This function suspends the calling thread until it receives a signal whose action is either to execute a signal-catching function or to terminate the process. The **spt_pause( )** function does not affect the action taken when the signal is received.

To catch externally-generated signals (such as SIGINT, SIGQUIT, SIGALRM, and SIGCHLD) at the thread level, you must export the SPT_THREAD_AWARE_SIGNAL environmental variable to the value 1. By default, SPT_THREAD_AWARE_SIGNAL is disabled.

## NOTES

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

When the received signal cause the calling process to terminate, the **spt_pause( )** function does not return a value. When the signal is caught by the calling thread and control is returned from the signal- catching function, the calling thread resumes execution from the point of suspension, the **spt_pause()** function returns the value **-**1, and **spt_pause()** function sets **errno** to the value [EINTR].

## ERRORS

If the following condition occurs, the **spt_pause( )** function sets **errno** to this value:

[EINTR] The signal was caught by the calling thread and control was returned from the signal-catching fucntion.

## RELATED INFORMATION

Functions: **pause(2)**, **pthread_kill(2)**, **pthread_sigmask(2)**, **sigsuspend(2) spt_sigaction(2)**, **spt_signal(2)**, **spt_sigsuspend(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_printf -** Initiates thread-aware **printf( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_printf(**
**const char ****format*, ...**);**

## PARAMETERS

See the **printf(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **printf( )** function. The file descriptor underlying standard output must be nonblocking for this function to be thread aware.

The following macro maps **spt_printf( )** to **printf( )** and has been defined in **spthread.h**:

**#define printf spt_printf**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **printf(3)** reference page. The following also applies:

- THe value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying standard output becomes invalid (is closed by another thread), -1 is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_printfx** - Prints formatted output to the standard output stream (thread-aware function)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

**int spt_printfx (**
    **const char ***format*
    [**,** *value*]. . .
    **);**

## PARAMETERS

*format* Specifies a character string combining literal characters with conversion specifications.

*value* Specifies the data to be converted according to the *format* parameter.

## DESCRIPTION

The **spt_printfx( )** function is the thread-aware version of the **printf( )** function.

The **spt_printfx( )** function converts, formats, and writes its *value* parameters, under control of the *format* parameter, to the standard output stream **stdout**.

The *format* parameter is a character string that contains two types of objects:

- Literal characters, which are copied to the output stream.
- Conversion specifications, each of which causes zero or more items to be fetched from the *value* parameter list.

If not enough items for *format* are in the *value* parameter list, the results are unpredictable. If more *value*s remain after the entire *format* has been processed, they are ignored.

### Conversion Specifications

Each conversion specification in the *format* parameter has the following syntax:

- A **%** (percent sign).

  The **spt_printfx( )** function can handle a format string that enables the system to process elements of the parameter list in variable order. In such a case, the normal conversion character **%** (percent sign) is replaced by **%***digit***$**, where *digit* is a decimal number in the range from 1 to **NL_ARGMAX**. Conversion is then applied to the specified argument, rather than to the next unused argument. This feature provides for the definition of format strings in an order appropriate to specific languages. When variable ordering is used, the ***** (asterisk) specification for field width in precision is replaced by **%***digit***$**. If the variable ordering feature is used, it must be specified for all conversions.

- Zero or more flags that modify the meaning of the conversion specification. The flag characters and their meanings are:

  **-** Left align the result of the conversion within the field.

**+** Begin the result of a signed conversion with a sign (+ or -).

(space) Prefix a space character to the result if the first character of a signed conversion is not a sign. If both the (space) and **+** flags appear, the (space) flag is ignored.

**#** Convert the value to an alternate form. For **o** conversion, it increases the precision to force the first digit of the result to be a 0 (zero). For **x** and **X** conversions, a nonzero result has 0x or 0X prefixed to it. For **e, E, f, g,** and **G** conversions, the result always contains a radix character, even if no digits follow it. For **g** and **G** conversions, trailing zeros are not removed from the result. For **c**, **C**, **d**, **i**, **s**, **S**, and **u** conversions, the flag has no effect.

**0** Pad to field width using leading zeros (following any indication of sign or base) for **d**, **e**, **E**, **f**, **g**,**G**, **i**, **o**, **u**, **x**, and **X** conversions; no space padding is performed. If the **0** and - (dash) flags both appear, the **0** flag will be ignored. For **d**, **i**, **o u**, **x**, and **X** conversions, if a precision is specified, the **0** flag is also ignored. For other conversions, the behavior is undefined.

- An optional decimal digit string that specifies the minimum field width. If the converted value has fewer characters than the field width, the field is padded on the left to the length specified by the field width. If the left-adjustment flag is specified, the field is padded on the right.

  A field width can be indicated by an * (asterisk) instead of a digit string. In this case, an integer (**int**) *value* parameter supplies the field width. The *value* parameter converted for output is not fetched until the conversion letter is reached, so the parameters specifying field width or precision must appear before the value (if any) to be converted. If the corresponding parameter has a negative value, it is treated as a **-** (dash) left alignment option followed by a positive field width. When variable ordering with the **L***digit***$** format is used, the ***** (asterisk) specification for field width in precision is replaced by *****digit***$**.

- An optional precision. The precision is a . (dot) followed by a decimal digit string. If no precision is given, it is treated as 0 (zero). The precision specifies:

  — The minimum number of digits to appear for the **d**, **u**, **o**, **x**, or **X** conversions.

  — The number of digits to appear after the radix character for the **e**, **E**, and **f** conversions.

  — The maximum number of significant digits for the **g** and **G** conversions.

  — The maximum number of bytes to be printed from a string in the **s** or **S** conversion.

  A field precision can be indicated by an * (asterisk) instead of a digit string. In this case, an integer (**int**) *value* parameter supplies the field precision. The *value* parameter converted for output is not fetched until the conversion letter is reached, so the parameters specifying field width or precision must appear before the value (if any) to be converted. If the value of the corresponding parameter is negative, it is treated as if the precision had not been specified. When variable ordering with the **L***digit***$** format is used, the ***** (asterisk) specification for field width in precision is replaced by *****digit***$**.

- An optional **h**, **l**, **ll**, or **L** indicating the size of the argument corresponding to the following integer or floating-point conversion specifier:
  - — An **h** followed by a **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier indicates that the argument will be treated as a **short int** or **unsigned short int**.
  - — An **h** followed by an **n** conversion specifier indicates that the argument will be treated as a pointer to a **short int**.
  - — An **l** followed by a **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier indicates that the argument will be treated as a **long int** or **unsigned long int**.
  - — An **l** followed by an **n** conversion specifier indicates that the argument will be treated as a pointer to a **long int**.
  - — An **ll** followed by a **d**, **i**, **o**, **u**, or **x** conversion code character indicates that the receiving variable is treated as a **long long int** or **unsigned long long int**.
  - — An **ll** followed by an **e**, **f**, or **g** conversion code character indicates that the receiving variable is treated as a **double** instead of a **float**.
  - — An **L** followed by a **e**, **E**, **f**, **g**, or **G** conversion specifier indicates that the argument will be treated as a **long double**.
  - — An **L** followed by a **d**, **i**, **o**, **x**, or **X** conversion specifier indicates that the argument will be treated as a **long long**, which is a 64-bit integer data type and an HP extension.
- A character that indicates the type of conversion to be applied:

**%** Performs no conversion. Prints **%**.

**d** or **i** Accepts an integer (**int**) *value* and converts it to signed decimal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a 0 (zero) value with a precision of 0 (zero) is a null string. Specifying a field width with a 0 (zero) as a leading character causes the field width value to be padded with leading zeros.

**u** Accepts an integer (**int**) *value* and converts it to unsigned decimal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a 0 (zero) value with a precision of 0 (zero) is a null string. Specifying a field width with a 0 (zero) as a leading character causes the field width value to be padded with leading zeros.

**o** Accepts an integer (**int**) *value* and converts it to unsigned octal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a 0 (zero) value with a precision of 0 (zero) is a null string. Specifying a field width with a 0 (zero) as a leading character causes the field width value to be padded with leading zeros. An octal value for field width is not implied.

**x** or **X** Accepts an integer (**int**) *value* and converts it to unsigned hexadecimal notation. The letters abcdef are used for the **x** conversion and the letters ABCDEF are used for the **X** conversion. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a 0 (zero) value with a precision of 0 (zero) is a null string. Specifying a field width with a 0 (zero) as a leading character causes the field width value to be padded with leading zeros.

**f** Accepts a **float** or **double** *value* and converts it to decimal notation in the format [**-**]*ddd.ddd*. The number of digits after the radix character is equal to the precision specification. If no precision is specified, six digits are output. If the precision is 0 (zero), no radix character appears (unless the **#** flag is specified). If a radix character is output, at least one digit is output before it. The value is rounded to the appropriate number of digits.

**e** or **E** Accepts a **float** or **double** *value* and converts it to the exponential form [-]*d.ddd***e**+/-*dd*. One digit is before the radix character and the number of digits after the readix character is equal to the precision specification. If no precision is specified, six digits are output. If the precision is 0 (zero), no radix character appears (unless the **#** flag is specified). The **E** conversion character produces a number with uppercase **E** instead of lowercase **e** before the exponent. The exponent always contains at least two digits. If the value is 0 (zero), the exponent is 0 (zero).

**g** or **G** Accepts a **float** or **double** *value* and converts it in the style of the **e**, **E**, or **f** conversion characters, with the precision specifying the number of significant digits. Trailing zeros are removed from the result. A radix character appears only if it is followed by a digit (except that it always appears if the **#** flag is specified). The style used depends on the value converted. Style **e** (**E**, if **G** is the flag used) results only if the exponent resulting from the conversion is less than -4, or if it is greater or equal to the precision.

**c** Accepts and prints an integer (**int**) *value* converted to an **unsigned char**.

**C** Accepts a **wchar_t** value, converts it to an array of bytes containing a multibyte character, and prints it. If a minimum field width is specified and the multibyte character occupies fewer bytes than the specified width, the multibyte character is padded with space characters to the specified width.

**s** Accepts a pointer to an array of **char** type. Bytes from the array are printed until a null character is encountered or the number of characters indicated by the precision is reached. If no precision is specified, all characters up to the first null character are printed. If the precision is not specified or is greater than the size of the array, the array must be terminated by a null byte. If the string pointer *value* has a value of 0 (zero) or null, the results are undefined.

**S** Accepts a pointer to an array of **wchar_t** type. Wide characters from the array are converted to an array of bytes containing multibyte characters and the multibyte characters up to (but not including) the null character are printed. If a precision is specified, no more than the number of bytes specified by the precision are printed. If the precision is not specified or is greater than the size of the array of bytes, the array of wide characters must be terminated by a null wide character. If a minimum field width is specified and the array of bytes occupy fewer bytes than the specified width, the array is padded with space characters to the specified width.

**p** Accepts a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, the same as unsigned hexadecimal integer (**x**).

**n** Accepts a pointer to an integer into which is written the number of characters written to the output stream so far by this call. No argument is converted.

If the result of a conversion is wider than the field width, the field is expanded to contain the converted result. No truncation occurs. However, a small precision can cause truncation on the right.

The **e**, **E**, **f**, and **g** formats represent the special floating-point values as follows:

Quiet NaN **NaN**

Signaling NaN **NaN**

+/-INF **+Inf** or **-Inf**

+/-0 +0.0 or -0.0 (zero)

The representation of the **+** (plus sign) depends on whether the **+** or (space) formatting flag is specified.

The **spt_printfx( )** function allows for the insertion of a language-dependent radix character in the output string. The radix character is defined by **langinfo** data in the program's locale (category **LC_NUMERIC**). In the C locale, or in a locale where the radix character is not defined, the radix character defaults to **.** (period).

The **st_ctime** and **st_mtime** fields of the file are marked for update between the successful execution of the **spt_printfx( )** function and the next successful completion of a call to the **spt_fflushx( )** or **spt_fclosex( )** functions on the same stream, or a call to the **exit( )** or **abort( )** functions.

## NOTES

The macro to map **printf( )** to **spt_printfx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **printf( )** to **spt_printfx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

This function supports both IEEE Std 754-1985 floating-point and Tandem floating-point values in the native environment. IEEE values can include NaN and infinity, and the sign of 0.0 (zero) can be either positive or negative. For a description of IEEE value classes, see the **fp_class(3)** reference page.

Guardian functions are available to convert between floating-point formats. For a discussion of floating-point conversions, see the *Guardian Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, this function returns the number of bytes in the output string. Otherwise, a negative value is returned.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

**ERRORS**

The **spt_printfx( )** function fails if either:

- The standard output stream is unbuffered
- The buffer for the standard output stream needs to be flushed and the function call causes an underlying **spt_writex( )** or **lseek( )** function to be invoked

In addition, if the **spt_printfx( )** function fails, **errno** is set to one of the following values:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying the output stream and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying the output stream is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EILSEQ] An invalid wide character was detected.

[EINTR] The operation was interrupted by a signal that was caught, and no data was transferred.

[EINVAL] There are insufficient arguments.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned. This error might also be returned under implementation-defined conditions.

[ENOMEM] Insufficient storage space was available.

[ENOSPC] No free space was remaining on the device containing the file.

[EBADF] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

## RELATED INFORMATION

Functions: **fp_class(3)**, **fprintf(3)**, **isnan(3)**, **toascii(3)**, **printf(3)**, **putc(3)**, **scanf(3)**, **sprintf(3)**, **spt_putcx(2)**, **spt_fprintfx(2)**, **spt_printf(2)**, **spt_sprintfx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_putc -** Initiates thread-aware **putc( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_putc(**
    **int** *c***,**
    **FILE ****stream***);**

## PARAMETERS

See the **putc(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **putc( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread-aware.

The following macro maps **spt_putc( )** to **putc( )** and has been defined in **spthread.h**:

**#define putc(c, stream) spt_putc(c, stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **putc(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying the stream becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_putchar -** Initiates thread-aware **putchar( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**extern int spt_putchar(**
**int** *c***);**

## PARAMETERS

See the **putchar(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **putchar( )** function. The file descriptor underlying standard output must be nonblocking for this function to be thread-aware.

The following macro maps **spt_fputchar( )** to **fptchar( )** and has been defined in **spthread.h**:

**#define putchar(c) spt_putchar(c)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **putchar(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying standard output becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_putcharx -** Writes a byte to the standard output stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

**int spt_putcharx (**
    **int** *cO*
    **);**

## PARAMETERS

*c* Specifies the character to be written.

## DESCRIPTION

The **spt_putcharx( )** function is the thread-aware version of the **putchar( )** function.

The **spt_putcharx( )** function writes the character *c* to the standard output stream. The character is written at the position at which the file pointer is currently pointing, if defined.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen( )** function causes it to become buffered or line buffered. Use the **setbuf( )** function to change the stream-buffering strategy.

When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as it is written. When an output stream is buffered, many characters are saved and written as a block. When an output stream is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a newline character is written or terminal input is requested).

The **st_ctime** and **st_mtime** fields of the file are marked for update between the successful execution of the **spt_putcharx( )** function, and the next successful completion of a call to the **spt_fflushx( )** or **spt_fclosex( )** function on the same stream, or a call to the **exit( )** or **abort( )** function.

## NOTES

The macro to map **putchar( )** to **spt_putcharx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **putchar( )** to **spt_putcharx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

The **spt_putcharx( )** function might be a macro (depending on the compile-time definitions used in the source). Consequently, you cannot use this interface where a function is necessary; for example, a subroutine pointer cannot point to it.

When a function is necessary, use the **spt_fputcx( )** function instead.

## RETURN VALUES

The **spt_putcharx( )** function and macro, upon successful completion, return the value written. If this function or macro fails, it returns the constant EOF. The function sets **errno** when an error is encountered.

If the file descriptor underlying **stdout** becomes invalid (is closed by another thread), **EOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **EOF** is returned with an **errno** value of [EINTR].

## ERRORS

The **spt_putcharx( )** function fails if:

- The standard output stream is not open for writing.
- The output file size cannot be increased.
- The standard output stream is unbuffered.
- The buffer of the standard output stream needs to be flushed and the function call causes an underlying **spt_writex( )** or **lseek( )** to be invoked and this underlying operation fails.

In addition, if any of these conditions occur, the **spt_putcharx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying the output stream and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying the output stream is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EINTR] The write operation was interrupted by a signal that was caught, and no data was transferred.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned. This error might also be returned under implementation-defined conditions.

[ENOMEM] Insufficient memory storage space is available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

Any error encountered during the underlying call to the **spt_writex( )** function can cause this function to return the corresponding **errno** value reported by the **spt_writex( )** function. If your application program encounters an **errno** value not listed on this reference page, see the the **spt_writex(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the cause of that error.

## RELATED INFORMATION

Functions: **ferror(3)**, **fputc(3)**, **getc(3)**, **getwc(3)**, **printf(3)**, **putc(3)**, **putchar(3)**, **puts(3)**, **putwc(3)**, **spt_fputcx(2)**, **spt_getcx(2)**, **spt_getwcx(2)**, **spt_printfx(2)**, **spt_putcx(2)**, **spt_putsx(2)**, **spt_putwcx(2)**, **spt_writex(2)**, **write(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_putcx -** Writes a byte to a specified output stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

**int spt_putcx (**
    **int** *c***,**
    **FILE ****stream*
    **);**

## PARAMETERS

*c* Specifies the character to be written.

*stream* Points to the file structure of an open file.

## DESCRIPTION

The **spt_putcx( )** function is the thread-aware version of the **putc( )** function.

The **spt_putcx( )** function writes the character *c* to the output specified by the *stream* parameter. The character is written at the position at which the file pointer is currently pointing, if defined.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen( )** function causes it to become buffered or line buffered. Use the **setbuf( )** function to change the stream buffering strategy.

When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as it is written. When an output stream is buffered, many characters are saved and written as a block. When an output stream is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a newline character is written or terminal input is requested).

The **st_ctime** and **st_mtime** fields of the file are marked for update between the successful execution of the **spt_putcx( )** function, and the next successful completion of a call to the **spt_fflushx( )** or **spt_fclosex( )** function on the same stream, or a call to the **exit( )** or **abort( )** function.

## NOTES

The macro to map **putc( )** to **spt_putcx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **putc( )** to **spt_putcx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

The **spt_putcx( )** function runs faster than **spt_fputcx( )**, but takes more space per invocation.

The **spt_putcx( )** function might be a macro (depending on the compile-time definitions used in the source). Consequently, you cannot use this interface where a function is necessary; for example, a subroutine pointer cannot point to it. In addition, **spt_putcx( )** does not work correctly with a *stream* parameter that has side effects. In particular, the following does not work:

**spt_putcx(*f++)**

When a function is necessary, use the **spt_fputcx( )** function instead.

**RETURN VALUES**

The **spt_putcx( )** function and macro, upon successful completion, returns the value written. If this function or macro fails, it returns the constant **EOF**. The function sets **errno** when an error is encountered.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), **EOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill(2)** function and is not blocked, ignored, or handled, **EOF** is returned with an **errno** value of [EINTR].

**ERRORS**

The **spt_putcx( )** function fails if:

- The *stream* parameter is not open for writing.
- The output file size cannot be increased.
- The *stream* is unbuffered.
- The buffer of the *stream* needs to be flushed and the function call causes an underlying **spt_writex( )** or **lseek( )** to be invoked and this underlying operation fails.

In addition, if any of these conditions occur, the **spt_putcx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying the output stream and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying the output stream is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EINTR] The write operation was interrupted by a signal that was caught, and no data was transferred.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned. This error might also be returned under implementation-defined conditions.

[ENOMEM] Insufficient memory storage space is available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

Any error encountered during the underlying call to the **spt_writex( )** function can cause this function to return the corresponding **errno** value reported by the **spt_writex( )** function. If your application program encounters an **errno** value not listed on this reference page, see the **spt_writex(2)** reference page either online or in the *Open System Services System Calls Reference Manual* for information about the cause of that error.

## RELATED INFORMATION

Functions: **ferror(3)**, **fputc(3)**, **getc(3)**, **getwc(3)**, **printf(3)**, **putc(3)**, **putchar(3)**, **puts(3)**, **putwc(3)**, **spt_fputcx(2)**, **spt_getcx(2)**, **spt_getwcx(2)**, **spt_printfx(2)**, **spt_putcharx(2)**, **spt_putwcx(2)**, **spt_writex(2)**, **write(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_puts -** Initiates thread-aware **puts( )** function.

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_puts(**
    **const char ***string*);

## PARAMETERS

See the **puts(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **puts( )** function. The file descriptor underlying standard output must be nonblocking for this function to be thread-aware.

The following macro maps **spt_puts( )** to **puts( )** and has been defined in **spthread.h**:

**#define puts(string) spt_puts(string)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **puts(3)** reference page. The following information also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying standard output becomes invalid (is closed by another thread), EOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, EOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_putsx -** Writes a string to the standard output stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
[#include <stdio.h>]
#include <spthread.h>

int spt_putsx (
        const char *string
        );
```

## PARAMETERS

*string* Points to a string to be written to output.

## DESCRIPTION

The **spt_putsx( )** function is the thread-aware version of the **puts( )** function.

The **spt_putsx( )** function writes the null-terminated string pointed to by the *string* parameter, followed by a newline character, to the standard output stream, **stdout**. This function does not write the terminating null byte.

The **st_ctime** and **st_mtime** fields of the file are marked for update between the successful execution of the **spt_putsx( )** function, and the next successful completion of a call to the **spt_fflushx( )** or **spt_fclosex( )** function on the same stream, or a call to the **exit( )** or **abort( )** function.

## NOTES

The macro to map **puts( )** to **spt_xputs( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **puts( )** to **spt_putsx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

Upon successful completion, the **spt_putsx( )** function returns the number of characters written. This function can return **EOF** on an error.

If the file descriptor underlying **stdout** becomes invalid (is closed by another thread), **EOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **EOF** is returned with an **errno** value of [EINTR].

## ERRORS

The **spt_putsx( )** function fails if either:

- The standard output stream is unbuffered.
- The buffer for the standard output stream needs to be flushed and the function call caused an underlying **spt_writex( )** or **lseek( )** to be invoked and this underlying operation fails with incomplete output.

In addition, if any of these conditions occur, the **spt_putsx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor of the underlying stream and the process would be delayed in the write operation.

[EBADF] The file descriptor of the underlying stream is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EINTR] The operation was interrupted by a signal that was caught, and no data was transferred.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**, and the process group of the process is orphaned. This error might also be returned under implementation-defined conditions.

[ENOMEM] Insufficient storage space available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

## RELATED INFORMATION

Functions: **fputs(3)**, **gets(3)**, **getws(3)**, **printf(3)**, **putc(3)**, **puts(3)**, **putwc(3)**, **putws(3)**, **spt_fputsx(2)**, **spt_getsx(2)**, **spt_getwsx(2)**, **spt_printfx(2)**, **spt_putcx(2)**, **spt_putwsx(2)**, **spt_writex(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_putw -** Initiates thread-aware **putw( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_putw(**
    **int** *c*,
    **FILE ****stream*);

## PARAMETERS

See the **putw(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **putw( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread-aware.

The following macro maps **spt_putw( )** to **putw( )** and has been defined in **spthread.h**:

**#define putw(c, stream) spt_putw(c, stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **putw(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying the stream becomes invalid (is closed by another thread), a nonzero value is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, a nonzero value is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_putwc -** Initiates thread-aware **putwc( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**wint_t spt_putwc(**
**wint_t** *c***,**
**FILE ****stream***);**

## PARAMETERS

See the **putwc(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **putwc( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread-aware.

The following macro maps **spt_putwc( )** to **putwc( )** and has been defined in **spthread.h**:

**#define putwc(c, stream) spt_putwc(c, stream)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **putwc(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying the stream becomes invalid (is closed by another thread), WEOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, WEOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_putwchar -** Initiates thread-aware **fputwchar( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

```
wint_t spt_putwchar(
        wint_t c);
```

## PARAMETERS

See the **putwchar(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **putwchar( )** function. The file descriptor underlying standard output must be non-locking for this function to be thread-aware.

The following macro maps **spt_putwchar( )** to **putwchar( )** and has been defined in **spthread.h**:

**#define putwchar(c) spt_putwchar(c)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **putwchar(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying standard output becomes invalid (is closed by another thread), WEOF is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, WEOF is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_putwcharx -** Writes a wide character to the standard output stream (thread-aware)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <wchar.h>**]
**#include <spthread.h>**

**wint_t spt_putwcharx (**
**wint_t** *c*
**);**

## PARAMETERS

*c* Specifies the wide character to be written.

## DESCRIPTION

The **spt_putwcharx( )** function is the thread-aware version of the **putwchar( )** function.

The **spt_putwcharx( )** function converts the **wchar_t** specified by the *c* parameter to its equivalent multibyte character and then writes the multibyte character to the standard output.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen( )** function causes it to become buffered or line buffered. Use the **setbuf( )** function to change the stream-buffering strategy.

## NOTES

The macro to map **putwchar( )** to **spt_putwcharx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **putwchar( )** to **spt_putwcharx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

Upon successful completion, this function returns the value written. If this function fails, it returns the constant **WEOF**.

If the file descriptor underlying **stdout** becomes invalid (is closed by another thread), **WEOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **WEOF** is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_putwcharx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying the standard output stream and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying the standard output stream is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EILSEQ] The wide character code specified by the *c* parameter does not correspond to a valid character.

[EINTR] The operation was interrupted by a signal that was caught, and no data was transferred.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned.

[ENOMEM] Insufficient storage space is available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

**RELATED INFORMATION**

Functions: **fputwc(3)**, **getc(3)**, **getwc(3)**, **printf(3)**, **putc(3)**, **puts(3)**, **putwc(3)**, **spt_fputwcx(2)**, **spt_getcx(2)**, **spt_getwcx(2)**, **spt_printfx(2)**, **spt_putcx(2)**, **spt_putsx(2)**, **spt_putwcx(2)**, **wctomb(3)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_putwcx -** Writes a wide character to a specified stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
[#include <wchar.h>]
#include <spthread.h>

wint_t spt_putwcx (
        wint_t c,
        FILE *stream
        );
```

## PARAMETERS

*c* Specifies the wide character to be written.

*stream* Points to the output data.

## DESCRIPTION

The **spt_putwcx( )** function is the thread-aware version of the **putwc( )** function.

The **spt_putwcx( )** function converts the **wchar_t** specified by the *c* parameter to its equivalent multibyte character and then writes the multibyte character to the *stream* parameter.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen( )** function causes it to become buffered or line buffered. Use the **setbuf( )** function to change the stream-buffering strategy.

## NOTES

The macro to map **putwc( )** to **spt_putwcx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **putwc( )** to **spt_putwcx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

Upon successful completion, this function returns the value written. If this function fails, it returns the constant **WEOF**.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), **WEOF** is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, **WEOF** is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_putwcx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EILSEQ] The wide character code specified by the *c* parameter does not correspond to a valid character.

[EINTR] The operation was interrupted by a signal that was caught, and no data was transferred.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned.

[ENOMEM] Insufficient storage space is available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

## RELATED INFORMATION

Functions: **fputwc(3)**, **getc(3)**, **getwc(3)**, **printf(3)**, **putc(3)**, **puts(3)**, **putwc(3)**, **spt_fputwcx(2)**, **spt_getcx(2)**, **spt_getwcx(2)**, **spt_printfx(2)**, **spt_putcx(2)**, **spt_putsx(2)**, **spt_putwcharx(2)**, **putwchar(3)**, **wctomb(3)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_putwx -** Writes a word to a stream (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <stdio.h>**]
**#include <spthread.h>**

**int spt_putwx (**
**int** *w***,**
**FILE ****stream*
**);**

## PARAMETERS

*stream* Points to the file structure of an open file.

*w* Specifies the word to be written.

## DESCRIPTION

The **spt_putwx( )** function is the thread-aware version of the **putw( )** function.

The **spt_putwx( )** function writes the word (**int**) specified by the *w* parameter to the output specified by the *stream* parameter. The word is written at the position at which the file pointer, if defined, is pointing. The size of a word is the size of an integer and varies from machine to machine. The **spt_putwx( )** function does not assume or cause special alignment of the data in the file.

Because of possible differences in word length and byte ordering, files written using the **spt_putwx( )** function are machine dependent, and might not be readable using the **spt_getwx( )** function on a different processor.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen( )** function causes it to become buffered or line buffered. Use the **setbuf( )** function to change the stream buffering strategy.

When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as it is written. When an output stream is buffered, many characters are saved and written as a block. When an output stream is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a newline character is written or terminal input is requested).

The **st_ctime** and **st_mtime** fields of the file are marked for update between the successful execution of the **spt_putwx( )**, function, and the next successful completion of a call to the **spt_fflushx( )** or **spt_fclosex( )** function on the same stream, or a call to the **exit( )** or **abort( )** function.

## NOTES

The macro to map **putw( )** to **spt_putwx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **putw( )** to **spt_putwx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## RETURN VALUES

The **spt_putwx( )** function, upon successful completion, returns a value of 0 (zero). Otherwise, it returns a nonzero value.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), a nonzero value is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, a nonzero value is returned with an **errno** value of [EINTR].

## ERRORS

The **spt_putwx( )** function fails if either:

- The *stream* is unbuffered
- The buffer of the *stream* needs to be flushed and the function call causes an underlying **spt_writex( )** or **lseek( )** to be invoked, and this underlying operation fails.

In addition, if any of the following conditions occur, the **spt_putwx( )** function sets **errno** to the corresponding value.

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EINTR] The write operation was interrupted by a signal that was caught, and no data was transferred.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned. This error might also be returned under implementation-defined conditions.

[ENOMEM] Insufficient storage space available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

## RELATED INFORMATION

Functions: **ferror(3)**, **fputc(3)**, **getc(3)**, **getwc(3)**, **printf(3)**, **putc(3)**, **putchar(3)**, **puts(3)**, **putwc(3)**, **spt_fputcx(2)**, **spt_getcx(2)**, **spt_getwcx(2)**, **spt_printfx(2)**, **spt_putcx(2)**, **spt_putcharx(2)**, **spt_putsx(2)**, **spt_putwcx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.**puts(3)**,

## NAME

**spt_read -** Initiates thread-aware **read( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**ssize_t spt_read(**
 **int** *filedes***,**
 **void ****buffer***,**
 **size_t** *nbytes***);**

## PARAMETERS

See the **read(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **read(** function. Note that file descriptor must be nonblocking for this function to be thread-aware.

For C applications, a macro to map **read( )** to **spt_read( )** is available when you use the **#define SPT_THREAD_AWARE** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **read( )** to **spt_read( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

## RETURN VALUES

See the **read(2)** reference page. The following also applies:

- The value of **errno** is never set to [EWOULDBLOCK] or [EAGAIN].
- If the file descriptor becomes invalid (for example, is closed by another thread), -1 is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_READLOCKX -** Sequentially locks and reads records in a Guardian disk file

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_READLOCKX (**
  **short** *filenum***,**
  **char ****buffer***,**
  **unsigned short** *read_count***,**
  [**unsigned short ****count_read* ]**,**
  [**long** *tag* ]
  **);**

## PARAMETERS

*filenum* specifies the file number of a Guardian file open instance that identifies the file to be read.

*buffer* specifies an array in the application process in which the information read from the file is returned.

*read_count* specifies the number of bytes to be read.

*count_read* is for waited I/O only. This parameter returns a count of the number of bytes returned from the file into *buffer*.

*tag* is for nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **SPT_READLOCKX( )** function is the thread-aware version of the Guardian READLOCKX procedure.

The **SPT_ READLOCKX( )** function sequentially locks and reads records in a Guardian disk file, exactly like the combination of an **SPT_LOCKREC( )** and **SPT_READX( )** call. **SPT_READLOCKX( )** is intended for use with 32-bit extended addresses. Therefore, the data buffer for **SPT_READLOCKX( )** can be either in the caller's stack segment or any extended data segment.

For programming information about the READLOCKX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

### Considerations

Nowait I/O and **SPT_READLOCKX( )**

If the **SPT_READLOCKX( )** function is used to initiate an operation with a file opened for nowait I/O, it must complete with a corresponding call to the Guardian AWAITIOX procedure.

Use for key-sequenced, relative, and entry-sequenced files

For key-sequenced, relative, and entry-sequenced files, a subset of the file (defined by the current access path, positioning mode, and comparison length) is locked and read with successive calls to **SPT_READLOCKX( )**.

For key-sequenced, relative, and entry-sequenced files, the first call to **SPT_READLOCKX( )** after a positioning (or open) locks and then returns the first record of the subset. Subsequent calls to **SPT_READLOCKX( )** without intermediate positioning locks returns successive records in the subset. After each of the subset's records are read, the position of the record just read becomes the file's current position. An attempt to read a record following the last record in a subset returns an EOF indication.

Locking records in an unstructured file

You can use **SPT_READLOCKX( )** to lock record positions, represented by a relative byte address (RBA), in an unstructured file. When sequentially reading an unstructured file with **SPT_READLOCKX( )**, each call to **SPT_READLOCK[X( )** first locks the RBA stored in the current next-record pointer and then returns record data beginning at that pointer for *read_count* bytes. After a successful call to **SPT_READLOCK[X( )**, the current-record pointer is set to the previous next-record pointer, and the next-record pointer is set to the previous next-record pointer plus *read_count*. This process repeats for each subsequent call to **SPT_READLOCKX( )**.

Location of *buffer* and *count_read*

The buffer and count transferred can be in the user stack or in an extended data segment. The *buffer* and *count_read* cannot be in the user code space.

If the *buffer* and *count_read* is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Transfer size

The size of the transfer is subject to current restrictions for the type of file.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **SPT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This restriction also applies to other processes that might be sharing the segment. It is the application's responsibility to ensure this.
- If you initiated the I/O with **SPT_READLOCKX( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- A selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.

- You can cancel Nowait I/O initiated with **SPT_READLOCKX( )** with a call to **SPT_CANCEL( )** or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or if the Guardian AWAITIOX procedure is called with a positive time limit and specific file number and the request times out.

Use of buffers
: A file opened by **SPT_FILE_OPEN_( )** uses direct I/O transfers by default; you can use **SPT_SETMODE(72)** to force the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers.

Bounds checking
: If the extended address of *buffer* is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

All considerations for the **SPT_READX( )** function also apply to this function.

**Use on OSS objects**

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **SPT_READLOCKX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

**NAME**

**SPT_READUPDATELOCKX -** Allows random processing of records in a Guardian disk file

**LIBRARY**

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

**SYNOPSIS**

```
[#include <cextdecs.h>]
#include <spthread.h>

short SPT_READUPDATELOCKX (
        short filenum,
        char *buffer,
        unsigned short read_count,
        [unsigned short *count_read ],
        [long tag ]
        );
```

**PARAMETERS**

*filenum* specifies the file number of a Guardian file open instance that identifies the file to be read.

*buffer* specifies an array in the application process in which the information read from the file is returned.

*read_count* specifies the number of bytes to be read.

*count_read* is for waited I/O only. This parameter returns a count of the number of bytes returned from the file into *buffer*.

*tag* is for nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

**DESCRIPTION**

The **SPT_READUPDATELOCKX()** function is the thread-aware version of the Guardian READUPDATELOCKX procedure.

You use **SPT_READUPDATELOCKX()** function for random processing of records in a Guardian disk file. This function first locks then reads the record from the current position in the file in anticipation of a subsequent call to the **SPT_WRITEUPDATEX()** or **SPT_WRITEUPDATEUNLOCK()** procedure. **SPT_READUPDATELOCKX()** is intended for reading a record after calling the Guardian POSITION or KEYPOSITION procedure.

**SPT_READUPDATELOCKX()** locks and reads the record in the same manner as the combination of the Guardian LOCKREC and READUPDATEX procedures but requires less system processing than the two separate calls would require.

For programming information about the READUPDATELOCKX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

**Considerations**

Buffer use **SPT_READUPDATELOCKX()** is intended for use with 32-bit extended addresses. Therefore, the data buffer for **SPT_READUPDATELOCKX()** can be either in the caller's stack segment or any extended data segment.

Nowait I/O and **SPT_READUPDATELOCKX( )**
The **SPT_READUPDATELOCKX( )** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

Use on nondisk files
If **SPT_READUPDATELOCKX( )** is performed on nondisk files, an error is returned.

Random processing
For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. Therefore, positioning for **SPT_READUPDATELOCKX( )** is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to **SPT_READUPDATELOCKX( )** is rejected with Guardian file-system error 11.

Queue files
To use **SPT_READUPDATELOCKX( )**, you must open a queue file with write access and with a *sync_or_receive_depth* of 0 (zero).

Location of *buffer* and *count_read*
The buffer and count transferred can be in the user stack or in an extended data segment. The *buffer* and *count_read* cannot be in the user code space.

If the *buffer* and *count_read* is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **SPT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This restriction also applies to other processes that might be sharing the segment. It is the application's responsibility to ensure this.
- If you initiated the I/O with **SPT_READUPDATELOCKX( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- A selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- You can cancel nowait I/O initiated with **SPT_READUPDATELOCKX( )** with a call to **SPT_CANCEL( )** or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or if the Guardian AWAITIOX procedure is called with a positive time limit and specific file number and the request times out.

Use of buffers
: A file opened by **SPT_FILE_OPEN_( )** uses direct I/O transfers by default; you can use **SPT_SETMODE(72)** to force the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers.

Bounds checking
: If the extended address of *buffer* is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

All considerations for the **SPT_LOCKREC( )** function also apply to this function. See also the "Disk File Considerations" for the Guardian READUPDATE procedure.

### Use on OSS objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **SPT_READUPDATELOCKX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_READUPDATEX -** Reads data from a Guardian disk or process file in anticipation of a subsequent write to the file

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_READUPDATEX (**
    **short** *filenum***,**
    **char ****buffer***,**
    **unsigned short** *read_count***,**
    [**unsigned short ****count_read* ]**,**
    [**long** *tag* ]
    **);**

## PARAMETERS

*filenum* specifies the file number of a Guardian file open instance that identifies the file to be read.

*buffer* specifies an array in the application process in which the information read from the file is returned.

*read_count* specifies the number of bytes to be read.

*count_read* is for waited I/O only. This parameter returns a count of the number of bytes returned from the file into *buffer*.

*tag* is for nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **SPT_READUPDATEX( )** function is the thread-aware version of the Guardian READUPDATEX procedure.

This function reads data from a disk or process file in anticipation of a subsequent write to the file. The values of the current-record and next-record pointers do not change. This function has the following uses:

Disk files **SPT_READUPDATEX( )** is used for random processing. Data is read from the file at the position of the current-record pointer. A call to this function typically follows a corresponding call to the Guardian POSITION or KEYPOSITION procedure.

Queue Files **SPT_READUPDATEX( )** is not supported on queue files. An attempt to use **SPT_READUPDATEX( )** is rejected with Guardian file-system error 2.

Interprocess communication
**SPT_READUPDATEX( )** reads a message from the $RECEIVE file that is answered in a later call to the Guardian REPLYX procedure. Each message read by **SPT_READUPDATEX( )** must be replied to in a corresponding call to REPLYX.

For programming information about the READUPDATEX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

**Considerations**

Buffer use
: **SPT_READUPDATEX( )** is intended for use with 32-bit extended addresses. Therefore, the data buffer for **SPT_READUPDATEX( )** can be either in the caller's stack segment or any extended data segment.

Random processing and positioning
: A call to **SPT_READUPDATEX( )** returns the record from the current position in the file. Because **SPT_READUPDATEX( )** is designed for random processing, it cannot be used for successive positioning through a subset of records as the **SPT_READX( )** function does. Rather, **SPT_READUPDATEX( )** reads a record after a call to the Guardian POSITION or KEYPOSITION procedure, in anticipation of a subsequent update through a call to the Guardian WRITEUPDATEX procedure.

Calling **SPT_READUPDATEX( )** after **SPT_READX( )**
: A call to **SPT_READUPDATEX( )** after a call to **SPT_READX( )**, without intermediate positioning, returns the same record as the call to **SPT_READX( )**.

Waited **SPT_READUPDATEX( )**
: If a waited **SPT_READUPDATEX( )** call is executed, the *count_read* parameter indicates the number of bytes actually read.

Nowait I/O and **SPT_READUPDATEX( )**
: If a nowait **SPT_READUPDATEX( )** call is executed, *count_read* has no meaning and can be omitted. The count of the number of bytes read is obtained when the I/O operation completes through the *count_transferred* parameter of the Guardian AWAITIOX procedure. The **SPT_READUPDATEX( )** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

Default locking mode action
: If the default locking mode is in effect when a call to **SPT_READUPDATEX( )** is made to a locked file or record, but the *filenum* of the locked file differs from the *filenum* in the call, the caller of **SPT_READUPDATEX( )** is suspended and queued in the locking queue behind other processes attempting to access the file or record.

**Use on OSS objects**

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **SPT_READUPDATEX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_readv -** Initiates thread-aware **readv( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**ssize_t spt_readv(**
**int** *filedes***,**
**struct iovec ****iov***,**
**int** *iov_count***);**

## PARAMETERS

See the **readv(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **readv( )** function. The file descriptor must be nonblocking for this function to be thread-aware.

For C applications, a macro to map **readv( )** to **spt_readv( )** is available when you use the **#define SPT_THREAD_AWARE** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **readv( )** to **spt_readv( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

## RETURN VALUES

See the **readv(2)** reference page. The following also applies:

- The value of **errno** is never set to [EWOULDBLOCK] or [EAGAIN].
- If the file descriptor becomes invalid (is closed by another thread), -1 is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_readvx -** Reads from a file into scattered buffers (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <sys/types.h>**]
[**#include <sys/uio.h>**]
**#include <spthread.h>**

**int spt_readvx (**
**int** *filedes***,**
**struct iovec ****iov***,**
**int** *iov_count*
**);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **spt_acceptx( )**, **creat( )**, **dup( )**, **spt_dup2x( )**, **spt_fcntlx( )**, **open( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*iov* Points to an **iovec** structure that identifies the buffers into which the data is to be placed.

*iov_count* Specifies the number of entries in the **iovec** structure pointed to by the *iov* parameter.

## DESCRIPTION

The **spt_readvx( )** function is a thread-aware version of the **readv( )** function.

The **spt_readvx( )** function attempts to read data from the file associated with the *filedes* parameter into a set of buffers. The **spt_readvx( )** function performs the same action as the **spt_readx( )** function, but it scatters the input data into the buffers specified by the array of **iovec** structure entries pointed to by the *iov* parameter.

On regular files and devices capable of seeking, the **spt_readvx( )** function starts at a position in the file given by the file pointer associated with the *filedes* parameter. Upon return from the **spt_readvx( )** function, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. For such devices, the value of the file pointer after a call to the **spt_readvx( )** function is always 0 (zero).

Upon successful completion, the **spt_readvx( )** function returns the number of bytes actually read and placed in the buffers.

No data transfer occurs past the current end-of-file (EOF). If the starting position is at or after the end-of-file, 0 (zero) is returned.

If an **spt_writex( )** or **spt_writevx( )** call contains so much data that the file system needs to resize a pipe or FIFO buffer, a read from that pipe or FIFO file can return up to 52 kilobytes of data, regardless of the size of **PIPE_BUF**. If the buffer cannot be resized for the write operation, a subsequent read from the pipe or FIFO file does not return more than 8192 bytes per call, regardless of the setting of **O_NONBLOCK**.

When attempting to read from an empty pipe (or FIFO file):

- If no process has the pipe open for writing, the **spt_readvx( )** function returns the value 0 (zero) to indicate EOF.

- If some process has the pipe open for writing:
  - If the **O_NONBLOCK** flag is not set, the **spt_readvx( )** function blocks until either some data is written or the pipe is closed by all processes that had opened the pipe for writing.
  - If the **O_NONBLOCK** flag is set, the **spt_readvx( )** function returns the value -1 and sets **errno** to [EAGAIN].

When attempting to read from a socket and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **spt_readvx( )** function blocks until data becomes available.
- If the **O_NONBLOCK** flag is set, the **spt_readvx( )** function returns the value -1 and sets **errno** to [EAGAIN]. The **O_NONBLOCK** flag has no effect if data is available.

When attempting to read from a character special file that supports nonblocking reads, such as a terminal, and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **spt_readvx( )** function blocks until data becomes available.
- If the **O_NONBLOCK** flag is set, the **spt_readvx( )** function returns the value -1 and sets **errno** to [EAGAIN]. The **O_NONBLOCK** flag has no effect if data is available.

If it is interrupted by a signal before it reads any data, the **spt_readvx( )** function returns the value -1 with **errno** set to [EINTR]. If it is interrupted by a signal after it has successfully read some data, the **spt_readvx( )** function returns the number of bytes read.

When reading from a device special file, the return of EOF has no effect on subsequent calls to the **spt_readvx( )** function. When modem disconnect is detected, an EOF is returned. The **errno** variable is not set to [EIO].

Upon successful completion, the **spt_readvx( )** function marks the **st_atime** field of the file for update.

The *iov_count* parameter specifies the number of entries (buffers) in the **iovec** structure pointed to by the *iov* parameter. Each **iovec** entry specifies the base address and length of an area in memory where data should be placed. The **spt_readvx( )** function always fills a buffer completely before proceeding to the next.

The **iovec** structure is defined in the **sys/uio.h** header file and contains entries with these members:

```
caddr_t   iov_base;
int       iov_len;
```

### Use on Guardian Objects

After a call to the **fork( )**, **tdm_fork( )**, or **tdm_spawn( )** function, the initial position within a Guardian EDIT file (a file in **/G** with file code 101) is the same for both parent and child processes. However, the position is not shared; moving the current position from within one process does not move it in the other process.

## NOTES

For C applications, a macro to map **readv( )** to **spt_readvx( )** is available when you use the **#define SPT_THREAD_AWARE_NONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **readv( )** to **spt_readvx( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

**RETURN VALUES**

Upon successful completion, the **spt_readvx( )** function returns the number of bytes actually read and placed into the buffers. The function guarantees to read the number of bytes requested only if the descriptor references a regular file that has at least that number of bytes left before EOF.

If a regular file does not contain enough bytes to satisfy the read or if the read otherwise fails, the value -1 is returned, **errno** is set to indicate the error, and the contents of the buffers are indeterminate.

If the file descriptor becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

**ERRORS**

If any of these conditions occur, the **spt_readvx( )** function sets **errno** to the corresponding value:

[EAGAIN] One of these conditions occurred:

- The **O_NONBLOCK** flag is set for the file descriptor, and the process would be delayed in the read operation.
- The **O_NONBLOCK** flag is set for the file descriptor, and no data was available.

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function (such as **spt_writez( )**) is in progress on a regular file and a function that is process-blocking for regular files (such as **read( )**, **spt_read( )**, or **spt_readx( )**) attempts to begin an I/O operation on the same open file.

[EBADF] The *filedes* parameter is not a valid file descriptor open for reading.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The **iov_base** member of the **iovec** structure points to a location outside of the allocated address space of the process.

[EFILEBAD] An attempt was made to read from a Guardian EDIT file (a file in **/G** with file code 101) with a corrupted internal structure.

[EINTR] An **spt_readvx( )** operation was interrupted by a signal before any data arrived.

[EINVAL] One of these conditions occurred:

- The sum of the **iov_len** values in the *iov* array was negative or overflowed a data item of type **ssize_t**.
- The value of the *iov_count* parameter was less than or equal to 0 (zero) or greater than **IOV_MAX**.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to read from its controlling terminal, the process is ignoring or blocking the **SIGTTIN** signal, or the process group is orphaned.
- A physical I/O error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed. Data might have been lost during a transfer.

[EISDIR] An **spt_readvx( )** operation was attempted against a directory.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote node, but communication with the remote node has been lost.

[ENOTCONN] The socket is no longer connected to a peer socket.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWRONGID] One of these conditions occurred:

- The process attempted an input or output operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for use of the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Functions: **creat(2)**, **dup(2)**, **fcntl(2)**, **ioctl(2)**, **lseek(2)**, **open(2)**, **opendir(3)**, **pipe(2)**, *readv(2)*, **socket(2)**. **socketpair(2)**, **spt_fcntlx(2)**, **spt_readv(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with these exceptions:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.
- When a signal arrives during a call to the **spt_readvx( )** function, instead of returning an EINTR error to the application, the **spt_readvx( )** retries the I/O operation, except in this case: If the **fork( )** function is called by a signal handler that is running on a thread performing an **spt_readvx( )** call, the **spt_readvx( )** call in the child process returns an EINTR error to the application.

## NAME

**spt_readvz -** Reads from a file into scattered buffers (thread-aware version)

## LIBRARY

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <spthread.h>

int spt_readvz(
        int filedes,
        struct iovec *iov,
        int iov_count);
```

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **spt_acceptx( )**, **creat( )**, **creat64( )**, **dup( )**, **spt_dup2x( )**, **spt_fcntlz( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*iov* Points to an **iovec** structure that identifies the buffers into which the data is to be placed.

*iov_count* Specifies the number of entries in the **iovec** structure pointed to by the *iov* parameter.

## DESCRIPTION

The **spt_readvz( )** function is a thread-aware version of the **readv( )** function.

The **spt_readvz( )** function attempts to read data from the file associated with the *filedes* parameter into a set of buffers. The **spt_readvz( )** function performs the same action as the **spt_readz( )** function, but it scatters the input data into the buffers specified by the array of **iovec** structure entries pointed to by the *iov* parameter.

On regular files and devices capable of seeking, the **spt_readvz( )** function starts at a position in the file given by the file pointer associated with the *filedes* parameter. Upon return from the **spt_readvz( )** function, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. For such devices, the value of the file pointer after a call to the **spt_readvz( )** function is always 0 (zero).

Upon successful completion, the **spt_readvz( )** function returns the number of bytes actually read and placed in the buffers.

No data transfer occurs past the current end-of-file (EOF). If the starting position is at or after the end-of-file, 0 (zero) is returned.

If an **spt_writez( )** or **spt_writevz( )** call contains so much data that the file system needs to resize a pipe or FIFO buffer, a read from that pipe or FIFO file can return up to 52 kilobytes of data, regardless of the size of **PIPE_BUF**. If the buffer cannot be resized for the write operation, a subsequent read from the pipe or FIFO file does not return more than 8192 bytes per call, regardless of the setting of **O_NONBLOCK**.

When attempting to read from an empty pipe (or FIFO file):

- If no process has the pipe open for writing, the **spt_readvz( )** function returns the value 0 (zero) to indicate EOF.
- If some process has the pipe open for writing:

— If the **O_NONBLOCK** flag is not set, the **spt_readvz( )** function blocks until either some data is written or the pipe is closed by all processes that had opened the pipe for writing.

— If the **O_NONBLOCK** flag is set, the **spt_readvz( )** function returns the value -1 and sets **errno** to [EAGAIN].

When attempting to read from a socket and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **spt_readvz( )** function blocks until data becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **spt_readvz( )** function returns the value -1 and sets **errno** to [EWOULDBLOCK].

When attempting to read from a character special file that supports nonblocking reads, such as a terminal, and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **spt_readvz( )** function blocks until data becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **spt_readvz( )** function returns the value -1 and sets **errno** to [EAGAIN].

If it is interrupted by a signal before it reads any data, the **spt_readvz( )** function returns the value -1 with **errno** set to [EINTR]. If it is interrupted by a signal after it has successfully read some data, the **spt_readvz( )** function returns the number of bytes read.

When reading from a device special file, the return of EOF has no effect on subsequent calls to the **spt_readvz( )** function. When modem disconnect is detected, an EOF is returned. The **errno** variable is not set to [EIO].

Upon successful completion, the **spt_readvz( )** function marks the **st_atime** field of the file for update.

The *iov_count* parameter specifies the number of entries (buffers) in the **iovec** structure pointed to by the *iov* parameter. Each **iovec** entry specifies the base address and length of an area in memory where data should be placed. The **spt_readvz( )** function always fills a buffer completely before proceeding to the next.

The **iovec** structure is defined in the **sys/uio.h** header file and contains entries with these members:

```
caddr_t   iov_base;
int       iov_len;
```

### Use on Guardian Objects

After a call to the **fork( )**, **tdm_fork( )**, or **tdm_spawn( )** function, the initial position within a Guardian EDIT file (a file in **/G** with file code 101) is the same for both parent and child processes. However, the position is not shared; moving the current position from within one process does not move it in the other process.

## NOTES

For file descriptors for non-regular files, the **spt_readvz( )** function behaves exactly the same as **spt_readvx( )**. For file descriptors for regular files, this is a thread-aware function: if this function must wait for an I/O operation to complete on an open file, this function blocks the thread that called it (instead of the entire process), while it waits for the I/O operation to complete.

This function serializes file operations on an open file. If a thread calls **spt_readvz( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **readv( )** to **spt_readvz( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **readv( )** to **spt_readvz( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

Upon successful completion, the **spt_readvz( )** function returns the number of bytes actually read and placed into the buffers. The function guarantees to read the number of bytes requested only if the descriptor references a regular file that has at least that number of bytes left before EOF.

If the read otherwise fails, the value -1 is returned, **errno** is set to indicate the error, and the contents of the buffers are indeterminate.

**ERRORS**

If any of these conditions occurs, the **spt_readvz( )** function sets **errno** to the corresponding value:

[EAGAIN] One of these conditions occurred:

- The **O_NONBLOCK** flag is set for the file descriptor, and the process would be delayed in the read operation.
- The **O_NONBLOCK** flag is set for the file descriptor, and no data was available.

[EBADF] The *filedes* parameter is not a valid file descriptor open for reading.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The **iov_base** memeber of the **iovec** structure points to a location outside of the allocated address space of the process.

[EFILEBAD] An attempt was made to read from a Guardian EDIT file (a file in **/G** with file code 101) with a corrupted internal structure.

[EINTR] A **spt_readvz( )** operation was interrupted by a signal before any data arrived.

[EINVAL] One of these conditions occurred:

- The sum of the **iov_len** values in the *iov* array was negative or overflowed a data item of type **ssize_t**.
- The value of the *iov_count* parameter was less than or equal to 0 (zero) or greater than **IOV_MAX**.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to read from its controlling terminal, the process is ignoring or blocking the **SIGTTIN** signal, or the process group is orphaned.
- A physical I/O error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed. Data might have been lost during a transfer.

[EISDIR] A **spt_readvz( )** operation was attempted against a directory.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOTCONN] The socket is no longer connected to a peer socket.

[EOVERFLOW]
The file is a regular file, the value of *nbyte* is greater than 0 (zero), the starting position is before the End-of-File (EOF), and the starting position is greater than or equal to the file offset maximum established when the file described by *filedes* was opened.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWOULDBLOCK]
The process attempted an operation on a socket for which **O_NONBLOCK** is set, there is no data, and no error has occurred.

[EWRONGID] One of these conditions occurred:

- The process attempted an input or output operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.

- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for use of the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **creat(2)**, **dup(2)**, **fcntl(2)**, **ioctl(2)**, **lseek(2)**, **open(2)**, **opendir(3)**, **pipe(2)**, **socket(2)**, **socketpair(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with these exceptions:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.
- When a signal arrives during a call to the **spt_readvz( )** function, instead of returning an EINTR error to the application, the **spt_readvz( )** retries the I/O operation, except in this case: If the **fork( )** function is called by a signal handler that is running on a thread performing an **spt_readvz( )** call, the **spt_readvz( )** call in the child process returns an EINTR error to the application.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [ECONNRESET], [EFAULT], [EFILEBAD], [EINVAL], [EISDIR], [EISGUARDIAN], [ENETDOWN], [ENOTCONN], [ETIMEDOUT], and [EWRONGID] can be returned.

## NAME

**SPT_READX -** Returns data from an open Guardian file to the application process data area

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_READX (**
  **short** *filenum***,**
  **char ****buffer***,**
  **unsigned short** *read_count***,**
  [**unsigned short ****count_read* ]**,**
  [**long** *tag* ]
  **);**

## PARAMETERS

*filenum* specifies the file number of a Guardian file open instance that identifies the file to be read.

*buffer* specifies an array in the application process in which the information read from the file is returned.

*read_count* specifies the number of bytes to be read.

*count_read* is for waited I/O only. This parameter returns a count of the number of bytes returned from the file into *buffer*.

*tag* is for nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **SPT_READX( )** function is the thread-aware version of the Guardian READX procedure.

The **SPT_READX( )** function returns data from an open Guardian file to the application process's data area. The **SPT_READX( )** function sequentially reads a disk file. For key-sequenced, relative, and entry-sequenced files, the **SPT_READX( )** function reads a subset of records in the file. (A subset of records is defined by an access path, positioning mode, and comparison length.)

For programming information about the Guardian READX file-system procedure, see the *Guardian Programmer's Guide*, the *Enscribe Programmer's Guide*, and the manuals for your specific data communications interface.

### General Considerations

Buffer use **SPT_READX( )** is intended for use with 32-bit extended addresses. Therefore, the data buffer for **SPT_READX( )** can be either in the caller's stack segment or any extended data segment.

Waited **SPT_READX( )**

If a waited **SPT_READX( )** call is executed, the *count_read* parameter indicates the number of bytes actually read.

Nowait **SPT_READX( )**

If a nowait **SPT_READX( )** call is executed, *count_read* has no meaning and can be omitted. The count of the number of bytes read is obtained through the *count-transferred* parameter of the Guardian AWAITIOX procedure when the I/O operation completes.

The **SPT_READX( )** function must complete with a call to the Guardian AWAITIOX procedure when it is used with a file that is opened for nowait I/O.

It is possible to initiate concurrent nowait read operations that share the same data buffer. To do this successfully with files opened by **SPT_FILE_OPEN_( )**, you must use **SPT_SETMODE( )** function 72 to cause the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers.

**SPT_READX( )** call when default locking mode is in effect

If the default locking mode is in effect when a call to **SPT_READX( )** is made to a locked file, but the *filenum* of the locked file differs from the *filenum* in the call, the caller of **SPT_READX( )** is suspended and queued in the locking queue behind other processes attempting to lock or read the file or record.

A deadlock condition occurs if a call to **SPT_READX( )** is made by a process having multiple opens on the same file and the *filenum* used to lock the file differs from the *filenum* supplied to **SPT_READX( )**.

Read call when alternate locking mode is in effect

If the alternate locking mode is in effect when **SPT_READX( )** is called, and the file or record is locked through a Guardian file number other than that supplied in the call, the call is rejected with Guardian file-system error 73 (`file is locked`).

Locking mode for read

The locking mode is specified by **SPT_SETMODE( )** function 4. If you encounter Guardian file-system error 73 (`file is locked`), you do not need to call **SPT_SETMODE( )** for every call to **SPT_READX( )**. **SPT_SETMODE())** stays in effect indefinitely (for example, until another **SPT_SETMODE( )** call is performed or the file is closed), and no additional overhead is involved.

Location of *buffer* and *count_read*

The buffer and count transferred can be in the user stack or in an extended data segment. The *buffer* and *count_read* cannot be in the user code space.

If the *buffer* and *count_read* are in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **SPT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This restriction also applies to other processes that might be sharing the segment. It is the application's responsibility to ensure this.

- If the I/O has been initiated with **SPT_READX( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- A selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- You can cancel nowait I/O initiated with **SPT_READX( )** with a call to **SPT_CANCEL( )** or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or if the Guardian AWAITIOX procedure is called with a positive time limit and specific file number and the request times out.

Use of buffers A file opened by **SPT_FILE_OPEN_( )** uses direct I/O transfers by default; you can use **SPT_SETMODE(72)** to force the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers.

Bounds checking
If the extended address of *buffer* is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

Queue files You can use **SPT_READX( )** to perform a nondestructive read of a queue file record. If the Guardian KEYPOSITIONX procedure is used to position to the beginning of the file, the first **SPT_READX( )** call performed returns a record with a length of 8 bytes and contents of all zeros. Subsequent **SPT_READX( )** calls return data from records written to the file.

**Disk File Considerations**

Large data transfers for unstructured files using default mode
For all read procedures, using default mode allows I/O sizes for unstructured files to be as large as 56 kilobytes (57,344), if the unstructured buffer size is 4 KB (4096). Default mode here refers to the mode of the file if **SPT_SETMODE( )** function 141 is not invoked.

For an unstructured file with an unstructured buffer size other than 4 KB, DP2 automatically adjusts the unstructured buffer size to 4 KB, if possible, when an I/O larger than 4KB is attempted. However, this adjustment is not possible for files that have extents with an odd number of pages; in such cases an I/O over 4 KB is not possible. The switch to a different unstructured buffer size will have a transient performance impact, so HP recommends that you set the size 4 KB initially, which is the default. Transfer sizes over 4 KB are not supported in default mode for unstructured access to structured files.

Large data transfers using **SPT_SETMODE(141)**
For **SPT_READX( )** only, large data transfers (more than 4096 bytes) can be done for unstructured access to structured or unstructured files, regardless of unstructured buffer size, by using **SPT_SETMODE( )** function 141. When you use **SPT_SETMODE(141)** to enable large data transfers, you can specify up to 56K (57344) bytes for the *read_count* parameter. For an explanation of function 141, see the Guardian SETMODE procedure description in the *Guardian Procedure Calls Reference Manual*.

Structured files

A subset of records for sequential **SPT_READX( )** calls
: The subset of records read by a series of calls to **SPT_READX( )** is specified through calls to the Guardian POSITION or KEYPOSITION procedures.

Reading of an approximate subset of records
: If an approximate subset is being read, the first record returned is the one whose key field, as indicated by the current key specifier, contains a value equal to or greater than the current key. Subsequent reading of the subset returns successive records until the last record in the file is read (an EOF indication is then returned).

Reading of a generic subset of records
: If a generic subset is being read, the first record returned is the one whose key field, as designated by the current-key specifier, contains a value equal to the current key for *comparison-length* bytes. Subsequent reading of the file returns successive records whose key matches the current key (for *comparison-length* bytes). When the current key no longer matches, an EOF indication returns.

  For relative and entry-sequenced files, a generic subset of the primary key is equivalent to an exact subset.

Reading of an exact subset of records
: If an exact subset is being read, the only records returned are those whose key field, as designated by the current-key specifier, contains a value of exactly the *comparison length* bytes (see the Guardian KEYPOSITION procedure in the *Guardian Procedure Calls Reference Manual*) and is equal to the key. When the current key no longer matches, an EOF indication returns. The exact subset for a key field having a unique value is at most one record.

Indicators after PT_READX( ) call
: After a successful **SPT_READX( )** call, the current-state indicators have these values:

  - Current position is the record just read.
  - Positioning mode is unchanged.
  - Comparison length is unchanged.
  - Current primary-key value is set to the value of the primary-key field in the record.

Unstructured files

Data transfer
: Data transfer begins from an unstructured disk file at the position indicated by the next-record pointer. The READ[X] procedure reads records sequentially on the basis of a beginning relative byte address (RBA) and the length of the records read.

Odd unstructured

If the unstructured file is created with the odd unstructured attribute (also known as ODDUNSTR) set, the number of bytes read is exactly the number of bytes specified with *read_count*. If the odd unstructured attribute is not set when the file is created, the value of *read_count* is rounded up to an even number before the **SPT_READX( )** operation is executed.

You set the odd unstructured attribute with the Guardian FILE_CREATE_, FILE_CREATELIST_, or CREATE procedure, or with the File Utility Program (FUP) SET and CREATE commands.

*read_count*

Unstructured files are transparently blocked. The BUFFERSIZE file attribute value, if not set by the user, defaults to 4096 bytes. The BUFFERSIZE attribute value (which is set by specifying **SPT_SETMODE( )** function 93) does not constrain the allowable *read_count* in any way. However, there is a performance penalty if the **SPT_READX( )** call does not start on a BUFFERSIZE boundary and does not have a *read_count* that is an integral multiple of the BUFFERSIZE. The DP2 disk process executes your requested I/O in (possibly multiple) units of BUFFERSIZE blocks starting on a block boundary.

*count_read* for unstructured reads

After a successful call to **SPT_READX( )** for an unstructured file, the value returned in *count_read* is the minimum of *read_count* or the EOF pointer minus the next-record pointer.

Pointers after an **SPT_READX( )** call

After a successful **SPT_READX( )** call to an unstructured file, the file pointers are:

- Current-record pointer is old next-record pointer.
- Next-record pointer is old next-record pointer plus *count_read*.

## RETURN VALUES

The **SPT_READX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_readx -** Reads from a file (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
[#include <sys/types.h>]
[#include <unistd.h>]
#include <spthread.h>

ssize_t spt_readx (
        int filedes,
        void *buffer,
        size_t nbytes
        );
```

## PARAMETERS

*filedes*
: Specifies an open file descriptor obtained from a successful call to the **spt_acceptx( )**, **creat( )**, **dup( )**, **spt_dup2x( )**, **spt_fcntlx( )**, **open( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*buffer*
: Points to the buffer to receive data read.

*nbytes*
: Specifies the number of bytes to read from the file associated with the *filedes* parameter.

  If the value of *nbytes* is 0 (zero), the **spt_readx( )** function returns 0 (zero). There are no other results.

  If the value of *nbytes* is greater than **SSIZE_MAX**, the **read( )** function returns -1 and sets **errno** to [EINVAL].

## DESCRIPTION

The **spt_readx( )** function is a thread-aware version of the **read( )** function.

The **spt_readx( )** function attempts to read *nbytes* bytes of data from the file associated with the *filedes* parameter into the buffer pointed to by the *buffer* parameter.

On regular files and devices capable of seeking, the **spt_readx( )** function starts at a position in the file given by the file pointer associated with the *filedes* parameter. Upon return from the **spt_readx( )** function, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. For such devices, the value of the file pointer after a call to the **spt_readx( )** function is always 0 (zero).

Upon successful completion, the **spt_readx( )** function returns the number of bytes actually read and placed in the buffer. This number is never greater than the value of the *nbytes* parameter.

The value returned can be less than *nbytes* if the number of bytes left in the file is less than *nbytes*, if the **spt_readx( )** request was interrupted by a signal, or if the file is a pipe, FIFO file, or special file and has fewer than *nbytes* bytes immediately available for reading. For example, an **spt_readx( )** from a file associated with a terminal might return one typed line of data.

No data transfer occurs past the current end-of-file (EOF). If the starting position is at or after the end-of-file, 0 (zero) is returned.

If an **spt_writex( )** or **spt_writevx( )** call contains so much data that the file system needs to resize a pipe or FIFO buffer, a read from that pipe or FIFO file can return up to 52 kilobytes of data, regardless of the size of **PIPE_BUF**. If the buffer cannot be resized for the write operation, a read from the pipe or FIFO file does not return more than 8192 bytes per call, regardless of the setting of **O_NONBLOCK**.

When attempting to read from an empty pipe (or FIFO file):

- If no process has the pipe open for writing, the **spt_readx()** function returns the value 0 (zero) to indicate EOF.
- If some process has the pipe open for writing:
  - — If the **O_NONBLOCK** flag is not set, the **spt_readx()** function blocks until either some data is written or the pipe is closed by all processes that had opened the pipe for writing.
  - — If the **O_NONBLOCK** flag is set, the **spt_readx()** function returns the value -1 and sets **errno** to [EAGAIN].

When attempting to read from a socket and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **spt_readx()** function blocks until data becomes available.
- If the **O_NONBLOCK** flag is set, the **spt_readx()** function returns the value -1 and sets **errno** to [EAGAIN]. The **O_NONBLOCK** flag has no effect if data is available.

When attempting to read from a character special file that supports nonblocking reads, such as a terminal, and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **spt_readx()** function blocks until data becomes available.
- If the **O_NONBLOCK** flag is set, the **spt_readx()** function returns the value -1 and sets **errno** to [EAGAIN]. The **O_NONBLOCK** flag has no effect if data is available.

If it is interrupted by a signal before it reads any data, the **read()** function returns the value -1 with **errno** set to [EINTR]. If it is interrupted by a signal after it has successfully read some data, the **read()** function returns the number of bytes read.

The **spt_readx()** function returns the number of bytes with the value 0 (zero) for any unwritten portion of a regular file before the EOF indication.

When reading from a device special file, the return of EOF has no effect on subsequent calls to the **spt_readx()** function. When modem disconnect is detected, an EOF indication is returned. The **errno** variable is not set to [EIO].

Upon successful completion, the **spt_readx()** function marks the **st_atime** field of the file for update.

### Use on Guardian Objects

After a call to the **fork()**, **tdm_fork()**, or **tdm_spawn()** function, the initial position within a Guardian EDIT file (a file in **/G** with file code 101) is the same for both parent and child processes. However, the position is not shared. Moving the current position from within one process does not move it in the other process.

## NOTES

For C applications, a macro to map **read()** to **spt_readx()** is available when you use the **#define SPT_THREAD_AWARE_NONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **read( )** to **spt_readx( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

**RETURN VALUES**

Upon successful completion, the **spt_readx( )** function returns the number of bytes actually read and placed into the buffer. The function guarantees to read the number of bytes requested only if the descriptor references a regular file that has at least that number of bytes left before EOF indication.

If a regular file does not contain enough bytes to satisfy the read, or if the read otherwise fails, the value -1 is returned, **errno** is set to indicate the error, and the contents of the buffer pointed to by the *buffer* parameter are indeterminate.

If the file descriptor becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

**ERRORS**

If any of these conditions occur, the **spt_readx( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor, and the process would be delayed in the read operation.

The **O_NONBLOCK** flag is set, and no data was available.

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function (such as **spt_writez( )**) is in progress on a regular file and a function that is process-blocking for regular files (such as **read( )**, **spt_read( )**, or **spt_readx( )**) attempts to begin an I/O operation on the same open file.

[EBADF] The *filedes* parameter is not a valid file descriptor open for reading.

[ECONNRESET]

One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The *buffer* parameter points to a location outside of the allocated address space of the process.

[EFILEBAD] An attempt was made to read from a Guardian EDIT file (a file in **/G** with file code 101) with a corrupted internal structure.

[EINTR] An **spt_readx( )** operation was interrupted by a signal before any data arrived.

[EINVAL] The value of the *nbytes* parameter is greater than **SSIZE_MAX**.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to read from its controlling terminal, the process is ignoring or blocking the **SIGTTIN** signal, or the process group is orphaned.

- A physical I/O error occurred. Data might have been lost during a transfer.

[EISDIR] An **spt_readx( )** operation was attempted against a directory.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote node, but communication with the remote node has been lost.

[ENOTCONN] The socket is no longer connected to a peer socket.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

### RELATED INFORMATION

Functions: **creat(2)**, **dup(2)**, **fcntl(2)**, **ioctl(2)**, **lseek(2)**, **open(2)**, **opendir(3)**, **pipe(2)**, **read(2)**, **socket(2)**, **spt_fcntlx(2)**, **spt_read(2)**.

### STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with these exceptions:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.
- When a signal arrives during a call to the **spt_readx( )** function, instead of returning an EINTR error to the application, the **spt_readx( )** retries the I/O operation, except in this case: If the **fork( )** function is called by a signal handler that is running on a thread performing an **spt_readx( )** call, the **spt_readx( )** call in the child process returns an EINTR error to the application.

## NAME

**spt_readz -** Reads from a file (thread-aware version)

## LIBRARY

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <sys/types.h>** /* optional except for POSIX.1 */
**#include <unistd.h>**
**#include <spthread.h>**

**ssize_t spt_readz(**
**int** *filedes***,**
**void** *****buffer***,**
**size_t** *nbytes***);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **spt_acceptx( )**, **creat( )**, **creat64( )**, **dup( )**, **spt_dup2x( )**, **spt_fcntlz( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*buffer* Points to the buffer to receive data read.

*nbytes* Specifies the number of bytes to read from the file associated with the *filedes* parameter.

If the value of *nbytes* is 0 (zero), the **spt_readz( )** function returns 0 (zero). There are no other results.

If the value of *nbytes* is greater than **SSIZE_MAX**, the **spt_readz( )** function returns -1 and sets **errno** to [EINVAL].

## DESCRIPTION

The **spt_readz( )** function is a thread-aware version of the **read( )** function for regular files and for special files.

The **spt_readz( )** function attempts to read *nbytes* bytes of data from the file associated with the *filedes* parameter into the buffer pointed to by the *buffer* parameter.

On regular files and devices capable of seeking, the **spt_readz( )** function starts at a position in the file given by the file pointer associated with the *filedes* parameter. Upon return from the **spt_readz( )** function, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. For such devices, the value of the file pointer after a call to the **spt_readz( )** function is always 0 (zero).

Upon successful completion, the **spt_readz( )** function returns the number of bytes actually read and placed in the buffer. This number is never greater than the value of the *nbytes* parameter.

The value returned can be less than *nbytes* if the number of bytes left in the file is less than *nbytes*, if the **spt_readz( )** request was interrupted by a signal, or if the file is a pipe, FIFO file, socket, or special file and has fewer than *nbytes* bytes immediately available for reading. For example, a **spt_readz( )** from a file associated with a terminal might return one typed line of data.

No data transfer occurs past the current end-of-file (EOF). If the starting position is at or after the end-of-file, 0 (zero) is returned.

If an **spt_writez( )** or **spt_writevz( )** call contains so much data that the file system needs to resize a pipe or FIFO buffer, a read from that pipe or FIFO file can return up to 52 kilobytes of data, regardless of the size of **PIPE_BUF**. If the buffer cannot be resized for the write operation, a read from the pipe or FIFO file does not return more than 8192 bytes per call, regardless of the setting of **O_NONBLOCK**.

When attempting to read from an empty pipe (or FIFO file):

- If no process has the pipe open for writing, the **spt_readz( )** function returns the value 0 (zero) to indicate EOF.
- If some process has the pipe open for writing:
  - — If the **O_NONBLOCK** flag is not set, the **spt_readz( )** function blocks until either some data is written or the pipe is closed by all processes that had opened the pipe for writing.
  - — If the **O_NONBLOCK** flag is set, the **spt_readz( )** function returns the value -1 and sets **errno** to [EAGAIN].

When attempting to read from a socket and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **spt_readz( )** function blocks until data becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **spt_readz( )** function returns the value -1 and sets **errno** to [EWOULDBLOCK].

When attempting to read from a character special file that supports nonblocking reads, such as a terminal, and no data is currently available:

- If the **O_NONBLOCK** flag is not set, the **spt_readz( )** function blocks until data becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **spt_readz( )** function returns the value -1 and sets **errno** to [EAGAIN].

If it is interrupted by a signal before it reads any data, the **spt_readz( )** function returns the value -1 with **errno** set to [EINTR]. If it is interrupted by a signal after it has successfully read some data, the **spt_readz( )** function returns the number of bytes read.

The **spt_readz( )** function returns the number of bytes with the value 0 (zero) for any unwritten portion of a regular file prior to EOF.

When reading from a device special file, the return of EOF has no effect on subsequent calls to the **spt_readz( )** function. When modem disconnect is detected, an EOF is returned. The **errno** variable is not set to [EIO].

Upon successful completion, the **spt_readz( )** function marks the **st_atime** field of the file for update.

### Use on Guardian Objects

After a call to the **fork( )**, **tdm_fork( )**, or **tdm_spawn( )** function, the initial position within a Guardian EDIT file (a file in **/G** with file code 101) is the same for both parent and child processes. However, the position is not shared. Moving the current position from within one process does not move it in the other process.

## NOTES

For file descriptors for special files, the **spt_readz( )** function behaves exactly the same as **spt_readx( )**. For file descriptors for regular files, this is a thread-aware function: if this function must wait for an I/O operation to complete on an open file, this function blocks the thread that called it (instead of the entire process), while it waits for the I/O operation to complete.

This function serializes file operations on an open file. If a thread calls **spt_readz( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **read( )** to **spt_readz( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **read( )** to **spt_readz( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

Upon successful completion, the **spt_readz( )** function returns the number of bytes actually read and placed into the buffer. The function guarantees to read the number of bytes requested only if the descriptor references a regular file that has at least that number of bytes left before EOF.

If the read otherwise fails, the value -1 is returned, **errno** is set to indicate the error, and the contents of the buffer pointed to by the *buffer* parameter are indeterminate.

## ERRORS

If any of these conditions occurs, the **spt_readz( )** function sets **errno** to the corresponding value:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor, and the process would be delayed in the read operation.

The **O_NONBLOCK** flag is set, and no data was available.

[EBADF] The *filedes* parameter is not a valid file descriptor open for reading.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The *buffer* parameter points to a location outside of the allocated address space of the process.

[EFILEBAD] An attempt was made to read from a Guardian EDIT file (a file in **/G** with file code 101) with a corrupted internal structure.

[EINTR] A **spt_readz( )** operation was interrupted by a signal before any data arrived.

[EINVAL] The value of the *nbytes* parameter is greater than **SSIZE_MAX**.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to read from its controlling terminal, the process is ignoring or blocking the **SIGTTIN** signal, or the process group is orphaned.
- A physical I/O error occurred. Data might have been lost during a transfer.

[EISDIR] A **spt_readz( )** operation was attempted against a directory.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOTCONN] The socket is no longer connected to a peer socket.

[EOVERFLOW]
The file is a regular file, the value of *nbyte* is greater than 0 (zero), the starting position is before the End-of-File (EOF), and the starting position is greater than or equal to the file offset maximum established when the file described by *filedes* was opened.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWOULDBLOCK]
The process attempted an operation on a socket for which **O_NONBLOCK** is set, there is no data, and no error has occurred.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **creat(2)**, **creat64(2)**, **dup(2)**, **fcntl(2)**, **ioctl(2)**, **lseek(2)**, **lseek64(2)**, **open(2)**, **open64(2)**, **opendir(3)**, **pipe(2)**, **socket(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with these exceptions:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.
- When a signal arrives during a call to the **spt_readz( )** function, instead of returning an EINTR error to the application, the **spt_readz( )** retries the I/O operation, except in this case: If the **fork( )** function is called by a signal handler that is running on a thread performing an **spt_readz( )** call, the **spt_readz( )** call in the child process returns an EINTR error to the application.

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- The value of the file pointer returned for a device that is incapable of seeking is always 0 (zero).
- When reading from a device special file, the return of EOF has no effect on subsequent calls to the **spt_readz( )** function.
- Specifying a value for the *nbytes* parameter that is greater than **SSIZE_MAX** causes the **spt_readz( )** function to return **-**1 and set **errno** to [EINVAL].
- **errno** can be set to [EIO] if a physical I/O error occurs.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [ECONNRESET], [EFAULT], [EFILEBAD], [EINVAL], [EISDIR], [EISGUARDIAN], [ENETDOWN], [ENOTCONN], [ETIMEDOUT], and [EWRONGID] can be returned.

**NAME**

**spt_RECEIVEREAD -** Initiates thread-aware function for reading $RECEIVE

**LIBRARY**

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

**SYNOPSIS**

**#include <spthread.h>**

**long spt_RECEIVEREAD(**
  **const short** *filenum***,**
  **char ****buffer***,**
  **const short** *read_count***,**
  **long ****count_read***,**
  **const long** *timelimit***,**
  **short ****receive_info***,**
  **short ****dialog_info***);**

**PARAMETERS**

*filenum* Specifies the Guardian file number for $RECEIVE (always 0)

*buffer* Specifies the data buffer

*read_count* Specifies the number of bytes to read

*count_read* Specifies the number of bytes read

*timelimit* Specifies a FILE_COMPLETE-style time limit

*receive_info* Specifies a FILE_GETRECEIVEINFO-style $RECEIVE info structure; NULL may be passed if this information is not needed; must not be NULL if *filenum*'s *receive_depth* is greater than 0 (zero).

*dialog_info* Specifies a FILE_GETRECEIVEINFO-style of dialog information (a **short int** used by context-sensitive Pathway servers); NULL can be passed if this information is not needed; NULL must be passed if *receive_info* is NULL.

**DESCRIPTION**

This thread-aware function is specifically for reading $RECEIVE. **spt_RECEIVEREAD( )** is slightly patterned after a combination of the READUPDATEX procedure and the FILE_GETRECEIVEINFO procedure, although its parameters do not match either of its modeled procedures. A side effect of calling **spt_RECEIVEREAD )*O puts the calling thread into a transaction (via a call to the SPT_TMF_SetTxHandle( )** function), if the received message was transactional. The calling thread may be blocked to honor the *filenum* value's receive depth. This allows any number of threads to simultaneously call **spt_RECEIVEREAD( )**. Blocked threads will be unblocked as other threads complete their calls to the **spt_REPLYX( )** function.

**NOTES**

Processing of the **spt_RECEIVEREAD( )** function cannot be interrupted by specifying **spt_interrupt(SPT_INTERRUPTED)**. The **spt_RECEIVEREAD( )** function responds to the attempt by retrying the input or output.

To interrupt the **spt_RECEIVEREAD( )** function, use one of the following function calls:

- **spt_wakeup(0, -1, 0,** *error***)** where *error* is any error number that can be recognized as a return value for the **spt_RECEIVEREAD( )** function.

- **spt_interrupt(0, SPT_ERROR)**.
- **spt_interrupt(0, SPT_TIMEDOUT)**.

Using any of these calls also cancels the input/output operation.

## RETURN VALUES

This function returns Guardian file-system error numbers including:

16 *filenum* is not registered.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_RECEIVEREADL** - Initiates thread-aware function for reading $RECEIVE (larger message version)

## LIBRARY

H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**long spt_RECEIVEREADL(**
**const short** *filenum***,**
**char ****buffer***,**
**const long** *read_count***,**
**long ****count_read***,**
**const long** *timelimit***,**
**short ****receive_info***,**

## PARAMETERS

*filenum* Specifies the Guardian file number for $RECEIVE (always 0)

*buffer* Specifies the data buffer

*read_count* Specifies the number of bytes to read

*count_read* Specifies the number of bytes read

*timelimit* Specifies a FILE_COMPLETEL_-style time limit

*receive_info* Specifies a FILE_GETRECEIVEINFOL_-style $RECEIVE info structure; NULL may be passed if this information is not needed; must not be NULL if *filenum*'s *receive_depth* is greater than 0 (zero).

## DESCRIPTION

This function is the same as the **spt_RECEIVEREAD( )** function, except that:

- This function can handle the longer message lengths allowed by the **SPT_SERVERCLASS_SENDL_( )** function.
- The *read_count* parameter is type **const long**.
- The *dialog_info* parameter is not included in the **spt_RECEIVEREADL( )** function.
- The Guardian file-system error 4184 (EVERSION) can be returned.

See the **spt_RECEIVEREAD(2)** reference page.

## NOTES

This function is supported on systems running J06.07 and later J-series RVUs and H06.18 and later H-series RVUs, and must be used instead of the **spt_RECEIVEREAD( )** function when the messages are larger than 32 kilobytes. This function also can be used for shorter messages.

## RETURN VALUES

See the **spt_RECEIVEREAD(2)** reference page.

In addition, this function can return this Guardian file-system error:

4184 (EVERSION)
The function was called from a system that is running a J-series RVU earlier than J06.07 or an H-series RVU earlier than H06.18.

## RELATED INFORMATION

Functions: **spt_RECEIVEREAD(2)**, **SPT_SERVERCLASS_SENDL_(3)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification.

## NAME

**spt_recv** - Initiates thread-aware **recv( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <spthread.h>**

**ssize_t spt_recv(**
  **int** *socket***,**
  **void ****buffer***,**
  **size_t** *length***,**
  **int** *flags***);**

## PARAMETERS

See the **recv(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **recv( )** function. The socket must be nonblocking for this function to be thread-aware.

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

The following macro maps **spt_recv( )** to **recv( )** and has been defined in **spthread.h**:

**#define recv(socket, buffer, length, flags)**
  **spt_recv(socket, buffer, length, flags)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **recv(2)** reference page. The following also applies:

- The value of **errno** is never set to [EWOULDBLOCK].
- If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_recvfrom -** Initiates thread-aware **recvfrom( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <spthread.h>**

**ssize_t spt_recvfrom(**
**int** *socket***,**
**void** ***buffer***,**
**size_t** *length***,**
**int** *flags***,**
**struct sockaddr** ***address***,**
**size_t** ***address_len***);**

## PARAMETERS

See the **recvfrom(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **recvfrom( )** function. The socket must be nonblocking for this function to be thread-aware.

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

The following macro maps **spt_recvfrom( )** to **recvfrom( )** and has been defined in **spthread.h**:

```
#define recvfrom(socket, buffer, length, flags, address, address_len)\
        spt_recvfrom(socket, buffer, length, flags, address, address_len)
```

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **recvfrom(2)** reference page. The following also applies:

- The value of **errno** is never set to [EWOULDBLOCK].
- If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** of [EINTR].

## ERRORS

See the **recvfrom(2)** reference page.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_recvfromx -** Receives a message from a socket (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <sys/socket.h>**]
**#include <spthread.h>**

**ssize_t spt_recvfromx(**
**int** *socket***,**
**void ****buffer***,**
**size_t** *length***,**
**int** *flags***,**
**struct sockaddr ****address***,**
**size_t ****address_len*
**);**

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*buffer* Points to the buffer where the message should be written.

*length* Specifies the length in bytes of the buffer pointed to by the *buffer* parameter.

*flags* Is a value that controls message reception. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_OOB** Requests out-of-band data.

**MSG_PEEK** Peeks at an incoming message. The data is treated as unread and the next call to the **spt_recvfromx( )** function (or similar function) will still return this data.

*address* Specifies either a null pointer or a pointer to a **sockaddr** structure in which the sending address is to be stored. The length and format of the address depend on the address family of the socket.

For **AF_INET** sockets, a pointer to the address structure **sockaddr_in** must be cast as a **struct sockaddr**. For **AF_INET6** sockets, a pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**. For **AF_UNIX** sockets, a pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*address_len* Points to a **size_t** data item, which, on input, specifies the length of the **sockaddr** structure that is pointed to by the *address* parameter, and, on return, specifies the length of the address stored.

## DESCRIPTION

The **spt_recvfromx( )** function is a thread-aware version of the **recvfrom( )** function.

The **spt_recvfromx( )** function receives messages from a connection-oriented or connectionless socket. **spt_recvfromx( )** is normally used with connectionless sockets because it includes parameters that permit a calling program to retrieve the source address of received data.

For message-based sockets (sockets of type **SOCK_DGRAM**), the entire message must be read in one call. If a message is too long to fit in the supplied buffer and **MSG_PEEK** is not set in the *flags* parameter, the excess bytes are discarded.

For stream-based sockets (sockets of type **SOCK_STREAM**), message boundaries are ignored. For such sockets, data is returned as soon as it becomes available; no data is discarded.

If no messages are available at the socket and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **spt_recvfromx( )** function blocks until a message arrives. If no messages are available at the socket and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **spt_recvfromx( )** function fails and sets **errno** to [EWOULD-BLOCK].

If the *address* parameter is not a null pointer, the source address of the received message is stored in the **sockaddr** structure pointed to by the *address* parameter, and the length of this address is stored in the object pointed to by the *address_len* parameter.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the address is truncated when stored.

**NOTES**

The macro to map **recvfrom( )** to **spt_recvfromx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **recvfrom( )** to **spt_recvfromx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

When data is available, a call to the **select( )** function indicates that the file descriptor for the socket is ready for reading.

**RETURN VALUES**

Upon successful completion, the **spt_recvfromx( )** function returns the length of the received message in bytes. If no data is available and the peer socket has performed an orderly shutdown, then 0 (zero) is returned.

If the **spt_recvfromx( )** function call fails, the value -1 is returned and **errno** is set to indicate the error.

If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

**ERRORS**

If any of these conditions occurs, the **spt_recvfromx( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINTR] A signal interrupted the function before any data was available.

[EINVAL] The **MSG_OOB** value is specified in the *flags* parameter and no out-of-band data is available.

[EIO] An input or output error occurred.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOTCONN] A receive operation was attempted on a connection-oriented socket that is not connected.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[ETIMEDOUT]
A transmission timed out on an active connection.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the operation would block.

**RELATED INFORMATION**

Functions: **fcntl(2)**, **read(2)**, **recv(2)**, **recvfrom(2)**, **recvmsg(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **spt_recvx(2)**, **spt_recvfrom(2)**, **spt_recvmsgx(2)**, **spt_sendtox(2)**, **spt_sendx(2)**, **spt_sendmsgx(2)**, **spt_writex(2)**, **write(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_recvmsg -** Initiates thread-aware **recvmsg(2)** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <spthread.h>**

**ssize_t spt_recvmsg(**
**int** *socket***,**
**struct msghdr ****message***,**
**int** *flags***);**

## PARAMETERS

See the **recvmsg(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **recvmsg( )** function. The socket must be nonblocking for this function to be thread-aware.

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

The following macro maps **spt_recvmsg( )** to **recvmsg( )** and has been defined in **spthread.h**:

```
#define recvmsg(socket, message, flags)\
        spt_recvmsg(socket, message, flags)
```

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE
```

See the **recvmsg(2)** reference page.

## RETURN VALUES

See the **recvmsg(2)** reference page. The following information also applies:

- The value of **errno** is never set to [EWOULDBLOCK].
- If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## ERRORS

See the **recvmsg(2)** reference page.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_recvmsgx -** Receives a message from a socket using a message structure (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <sys/socket.h>**]
**#include <spthread.h>**

**ssize_t spt_recvmsgx(**
  **int** *socket***,**
  **struct msghdr ****message***,**
  **int** *flags*
  **);**

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*message* Points to a **msghdr** structure containing both the buffer to store the source address and the buffers for the incoming message. The length and format of the address depend on the address family for the socket. The **msg_flags** member of the structure is ignored on input but might contain meaningful values on output. For:

**AF_INET** sockets
A pointer in **msghdr** to the address structure **sockaddr_in** must be cast as a **struct sockaddr**.

**AF_INET6** sockets
A pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**.

**AF_UNIX** sockets
A pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*flags* Is a value that controls message reception. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_OOB** Requests out-of-band data.

**MSG_PEEK** Peeks at an incoming message. The data is treated as unread, and the next call to the **spt_recvmsgx( )** function (or a similar function) will still return this data.

## DESCRIPTION

The **spt_recvmsgx( )** function is a thread-aware version of the **recvmsg( )** function.

The **spt_recvmsgx( )** function receives messages from a connection-oriented or connectionless socket using the **msghdr** structure. The **spt_recvmsgx( )** function is normally used with connectionless sockets because it includes parameters that permit a calling program to retrieve the source address of the received data.

For message-based sockets (sockets of type **SOCK_DGRAM**), the entire message must be read in one call. If a message is too long to fit in the supplied buffer and **MSG_PEEK** is not set in the *flags* parameter, the excess bytes are discarded, and **MSG_TRUNC** is set in the **msg_flags** field of the **msghdr** structure.

For stream-based sockets (sockets of type **SOCK_STREAM**), message boundaries are ignored. For such sockets, data is returned as soon as it becomes available; no data is discarded.

If no messages are available at the socket and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **spt_recvmsgx( )** function blocks until a message arrives. If no messages are available at the socket and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **spt_msgx( )** function fails and sets **errno** to [EWOULDBLOCK].

In the **msghdr** structure, the **msg_name** and **msg_namelen** members specify the source address if the socket is unconnected. If the socket is connected, the **msg_name** and **msg_namelen** members are ignored. The **msg_name** member can be a null pointer if no names are desired or required. The **msg_iov** and **msg_iovlen** members describe the scatter and gather locations.

Upon successful completion of the **spt_recvmsgx( )** call, the value of the **msg_flags** member of the **msghdr** structure is the bitwise OR of zero or more of the following values:

**MSG_CTRUNC**
Control data was truncated.

**MSG_OOB** Out-of-band data was received.

**MSG_TRUNC**
Normal data was truncated.

In the **msghdr** structure, the **msg_control** and **msg_controllen** members specify the ancillary data buffer that only sockets in the **AF_UNIX** domain can use to receive file descriptors passed from another process on the same node. The **msg_control** member can be a null pointer if ancillary data is not desired or required. If the **msg_control** member is nonnull, on input the **msg_controllen** member contains the size of the ancillary data buffer and on output it contains the size of the received ancillary data. If, on output, the **msg_controllen** member is nonzero, the ancillary data buffer contains a **cmsghdr** structure followed by 1 to 16 file descriptors.

If **spt_recvmsgx( )** is called with an ancillary data buffer and **MSG_PEEK** is set, the **msg_controllen** member is valid, but the ancillary data is not meaningful (no file descriptors are received). Ancillary data is not discarded but remains available for the next call to **spt_recvmsgx( )** where **MSG_PEEK** is set.

If **spt_recvmsgx( )** is called with an ancillary data buffer that is too small to hold the available file descriptors, **MSG_CTRUNC** is set, and the excess file descriptors are discarded.

If **spt_recvmsgx( )** is called with an ancillary data buffer and one or more of the received file descriptors are unusable (perhaps because of a device error), no error is indicated until the file descriptor is used.

## NOTES

The macro to map **recvmsg( )** to **spt_recvmsgx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **recvmsg( )** to **spt_recvmsgx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the

following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

When data is available, a call to the **select( )** function indicates that the file descriptor for the socket is ready for reading.

**RETURN VALUES**

Upon successful completion, the **spt_recvmsgx( )** function returns the length of the received message in bytes. If no data is available and the peer socket has performed an orderly shutdown, 0 (zero) is returned.

If the **spt_recvmsgx( )** function call fails, the value -1 is returned, and **errno** is set to indicate the error.

If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

**ERRORS**

If any of these conditions occur, the **spt_recvmsgx( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINTR] A signal interrupted the function before any data was available.

[EINVAL] One of these conditions occurred:

- The **MSG_OOB** value is specified in the *flags* parameter, and no out-of-band data is available.
- The sum of the values specified for the **msg_iovlen** field of the **msghdr** structure is too large for a data item of type **ssize_t**.
- The socket belongs to the **AF_INET** or **AF_INET6** domain, and the function call requested **msg_control** data.
- The socket belongs to the **AF_UNIX** domain, and the size of **msg_controllen** is less than the size of the **cmsghdr** structure plus one file descriptor.

[EIO] An input or output error occurred.

[EMFILE] The socket is in the **AF_UNIX** domain, and processing the **cmsghdr** structure would cause the receiving process to exceed **OPEN_MAX**.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOTCONN] A receive operation was attempted on a connection-oriented socket that is not connected.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
A specified value for the *flags* parameter is not supported for this socket type.

[ETIMEDOUT]
A transmission timed out on an active connection.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set), and the operation would block.

**RELATED INFORMATION**

Functions: **fcntl(2)**, **recv(2)**, **recvfrom(2)**, **recvmsg(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **socketpair(2)**, **spt_recvx(2)**, **spt_recvfromx(2)**, **spt_recvmsg(2)**, **spt_sendtox(2)**, **spt_sendx(2)**, **spt_sendmsgx(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_recvx -** Receives a message from a connected socket (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <sys/socket.h>**]
**#include <spthread.h>**

**ssize_t spt_recvx(**
 **int** *socket***,**
 **void** ***buffer***,**
 **size_t** *length***,**
 **int** *flags*
 **);**

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*buffer* Points to the buffer where the message should be written.

*length* Specifies the length in bytes of the buffer pointed to by the *buffer* parameter.

*flags* Is a value that controls message reception. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_OOB** Requests out-of-band data.

**MSG_PEEK** Peeks at an incoming message. The data is treated as unread and the next call to the **spt_recvx( )** function (or similar function) will still return this data.

## DESCRIPTION

The **spt_recvx( )** function is a thread-aware version of the **recv( )** function.

The **spt_recvx( )** function receives messages from a connected socket.

For message-based sockets (sockets of type **SOCK_DGRAM**), the entire message must be read in one call. If a message is too long to fit in the supplied buffer and **MSG_PEEK** is not set in the *flags* parameter, the excess bytes are discarded.

For stream-based sockets (sockets of type **SOCK_STREAM**), message boundaries are ignored. For such sockets, data is returned as soon as it becomes available; no data is discarded.

If no messages are available at the socket and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **spt_recvx( )** function blocks until a message arrives. If no messages are available at the socket and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **spt_recvx( )** function fails and sets **errno** to [EWOULDBLOCK].

## NOTES

The macro to map **recv( )** to **spt_recvx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **recv( )** to **spt_recvx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

When data is available, a call to the **select( )** function indicates that the file descriptor for the socket is ready for reading.

Calling the **spt_recvx( )** function with a *flags* parameter of 0 (zero) is identical to calling the **spt_readx( )** function.

**RETURN VALUES**

Upon successful completion, the **spt_recvx( )** function returns the length of the received message in bytes. If no data is available and the peer socket has performed an orderly shutdown, then 0 (zero) is returned.

If the **spt_recvx( )** function call fails, the value -1 is returned and **errno** is set to indicate the error.

If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

**ERRORS**

If any of these conditions occur, the **spt_recvx( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EFAULT] A user-supplied memory buffer cannot be accessed or written.

[EINTR] A signal interrupted the function before any data was available.

[EINVAL] The **MSG_OOB** value is specified in the *flags* parameter and no out-of-band data is available.

[EIO] An input or output error occurred.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOTCONN] A receive operation was attempted on a connection-oriented socket that is not connected.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[ETIMEDOUT]
A transmission timed out on an active connection.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the operation would block.

**RELATED INFORMATION**

Functions: **fcntl(2)**, **read(2)**, **recv(2)**, **recvfrom(2)**, **recvmsg(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **spt_readx(2)**, **spt_recv(2)**, **spt_recvfromx(2)**, **spt_recvmsgx(2)**, **spt_sendtox(2)**, **spt_sendx(2)**, **spt_sendmsgx(2)**, **spt_writex(2)**, **write(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_regFile** - Registers the file number

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**spt_error_t spt_regFile(**
**const short** *filenum***);**

## PARAMETERS

*filenum* Specifies the Guardian file number of the file being registered

## DESCRIPTION

Registers the file number as one that the user will manage through the default callback.

## RETURN VALUES

See the **spt_regFileIOHandler(2)** reference page.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_regFileIOHandler -** Registers the file number

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**spt_error_t spt_regFileIOHandler(**
    **const short** *filenum***,**
    **const spt_FileIOHandler_p** *functionPtr***);**

## PARAMETERS

*filenum* Specifies the Guardian file number for the file being registered

*functionPtr* Specifies user-supplied callback. This function must not block its invoking thread; for example, it should not call the **spt_awaitio( )** function

## DESCRIPTION

This function registers the file number as one that the user will manage through a user-supplied callback. This callback is invoked immediately after each I/O on *filenum* completes.

## RETURN VALUES

**SPT_SUCCESS**
THe Guardian file number was successfully registered.

**SPT_ERROR** The value specified for *filenum* was less than 0 (zero).

**SPT_ERROR** *filenum* was already registered prior to this call.

**SPT_ERROR** The FILE_COMPLETE_SET_ procedure addition of *filenum* returned a nonzero value.

**SPT_ERROR** *functionPtr* is NULL.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_regOSSFileIOHandler -** Registers the file descriptor to manage through a callback function

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

```
#include <spthread.h>

spt_error_t spt_regOSSFileIOHandler(
        const int filedes,
        const spt_OSSFileIOHandler_p functionPtr);
```

## PARAMETERS

*filedes* Specifies the OSS file descriptor being registered

*functionPtr* Specifies the user-supplied callback function; this function must not block

## DESCRIPTION

This function registers the file descriptor as one that the user will manage through a user-supplied callback.

## RETURN VALUES

**SPT_SUCCESS**
Value for file descriptor was registered.

**SPT_ERROR** The specified *filedes* was less than 0 (zero).

**SPT_ERROR** *filedes* was already registered prior to this call.

**SPT_ERROR** *functionPtr* is NULL.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_regPathsendFile -** Registers the Pathsend file number

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**spt_error_t spt_regPathsendFile(**
**const short** *fileno* **);**

## PARAMETERS

*fileno* Contains the *scsend-op-num* value obtained during the first nowaited SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ procedure call.

## DESCRIPTION

This function is used to register the Pathsend file number. This function should be called immediately after the first call to a SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ procedure call.

## RETURN VALUES

**SPT_SUCCESS**
The Pathsend file number was successfully registered.

**SPT_ERROR** The specified Pathsend file number is already registered.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_regPathsendTagHandler -** Registers the user-supplied Pathsend tag

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**spt_error_t spt_regPathsendTagHandler(**
**const long** *tag***,**
**spt_FileIOHandler_p** *callback***,**
**void *** *userdata* **);**

## PARAMETERS

*tag* Specifies the Pathsend tag that should be registered.

*callback* Specifies a user-supplied callback function. This function should not block its invoking thread. The callback function should have the following prototype:

```
callback(const short filenum,
                /* Guardian file number
                  being waited on */
         const long tag,
                /* tag being waited on or
                  -1 for all tags */
         const long completionCount,
                /* byte transfer count
                  of completed IO */
         const long fserror,
                /* Guardian error number for IO */
         void * userdata
                /* for communication between
                  I/O initiator and callback. */
         );
```

*userdata* Specifies data to be communicated between the I/O initiator and the callback function.

## DESCRIPTION

This function registers the Pathsend tag as a tag that the user will manage through a user-supplied callback function. The callback function is invoked when a Pathsend operation that uses the tag completes.

## RETURN VALUES

**SPT_SUCCESS**
The specified tag was registered.

**SPT_ERROR** Another Pathsend handler has already registered the tag.

## RELATED INFORMATION

Functions: **spt_unregPathsendTagHandler(2)**, **SPT_SERVERCLASS_DIALOG_ABORT_(2)**, **SPT_SERVERCLASS_DIALOG_BEGIN_(2)**, **SPT_SERVERCLASS_DIALOG_END_(2)**, **SPT_SERVERCLASS_DIALOG_SEND_(2)**, **SPT_SERVERCLASS_SEND_INFO_(2)**, **SPT_SERVERCLASS_SEND_(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_regTimerHandler** - Registers a user-supplied timer callback function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**spt_error_t spt_regTimerHandler(**
    **const spt_TimerHandler_p** *functionPtr***);**

## PARAMETERS

*functionPtr* Specifies the user-supplied callback function; this function must not block I/O

## DESCRIPTION

This function registers a user-supplied timer callback function.

## RETURN VALUES

**SPT_SUCCESS**
The callback function was successfully registered.

**SPT_ERROR** *functionPtr* is NULL.

**SPT_ERROR** The specified callback function is already registered.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_REPLYX -** Initiates thread-aware REPLYX procedure call

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**long spt_REPLYX(**
  **const char ****buffer***,**
  **const short** *write_count***,**
  **short ****count_written***,**
  **const short** *msg_tag***,**
  **const short** *error_return***);**

## PARAMETERS

*buffer* Specifies data buffer

*write_count* Specifies the number of bytes to write

*count_written* Specifies the number of bytes written; might be NULL

*msg_tag* Specifies required tag identifying message to reply to and is ignored if the corresponding Guardian file number receive depth is 1

*error_return* Specifies a Guardian file-system error to return to sender

## DESCRIPTION

This is a thread-aware version of the REPLYX procedure call; this function clears the thread's transaction context if appropriate.

## RETURN VALUES

This function returns a Guardian file-system error number.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_REPLYXL -** Initiates thread-aware REPLYXL procedure call (larger message version)

## LIBRARY

H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**long spt_REPLYXL(**
    **const char ****buffer***,**
    **const long** *write_count***,**
    **long ****count_written***,**
    **const short** *msg_tag***,**
    **const short** *error_return***);**

## PARAMETERS

*buffer* Specifies data buffer

*write_count* Specifies the number of bytes to write

*count_written* Specifies the number of bytes written; might be NULL

*msg_tag* Specifies required tag identifying message to reply to and is ignored if the corresponding Guardian file number receive depth is 1

*error_return* Specifies a Guardian file-system error to return to sender

## DESCRIPTION

This function is the same as the **spt_REPLYX( )** function, except:

- This function can handle the longer message lengths allowed by the **SPT_SERVERCLASS_SENDL_( )** function.
- The *write_count* parameter is type **const long**.
- The *count_written* parameter is type **long**.
- The Guardian file-system error 4184 (EVERSION) can be returned.

See the **spt_REPLYX(2)** reference page.

## NOTES

This function is supported on systems running J06.07 and later J-series RVUs and H06.18 and later H-series RVUs, and must be used instead of the **spt_REPLYX( )** function when the messages are larger than 32 kilobytes long. This function also can be used for shorter messages.

## RETURN VALUES

See the **spt_REPLYX(2)** reference page.

In addition, this function can return this Guardian file-system error:

4184 (EVERSION)
The function was called from a system that is running a J-series RVU earlier than to J06.07 or an H-series RVU earlier than H06.18.

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 Version 2 specification.

## NAME

**spt_select -** Initiates thread-aware **select( )** function for mulitple file descriptors

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_select(**
    **int** *nfds***,**
    **fd_set ****readfds***,**
    **fd_set ****writefds***,**
    **fd_set ****errorfds***,**
    **struct timeval ****timeout***);**

## PARAMETERS

See the **select(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **select( )** function that is used to check the status of multiple file descriptors. To check the status of a single file descriptor, use the **spt_select_single_np( )** function, which provides better performance.

In **spthread.h**, a mapping of **select( )** to **spt_select( )** has been defined:

**#define select(nfds, readfds, writefds, errorfds, timeout)**
    **spt_select(nfds, readfds, writefds, errorfds, timeout)**

For C applications that do not use the nonblocking feature, this mapping is available only when the correct preprocessor has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**
**#include <spthread.h>**

For C applications that use the nonblocking feature, this mapping is available only when the correct preprocessor has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE_NONBLOCK**
**#include <spthread.h>**

For C++ applications that do not use the nonblocking feature, this mapping is available only when the correct preprocessor has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE_PRAGMA**
**#include <spthread.h>**

For C++ applications that use the nonblocking feature, this mapping is available only when the correct preprocessor has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**
**#include <spthread.h>**

## NOTES

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.

- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

To use a combination of the **spt_select( )** and the **spt_select_single_np( )** functions in a single source file, you must explicitly call these functions.

**RETURN VALUES**

See the **select(2)** reference page. The following information also applies:

- If the file descriptor becomes invalid (is closed by another thread), -1 is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

**RELATED INFORMATION**

Functions: **select(2)**, **spt_select_single_np(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_select_single_np -** Initiates thread-aware **select( )** function for a single file descriptor

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

```
#include <spthread.h>

int spt_select_single_np(
        int nfds,
        fd_set *readfds,
        fd_set *writefds,
        fd_set *errorfds,
        struct timeval *timeout);
```

## PARAMETERS

See the **select(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **select( )** function used to check the status of a single file descriptor. To improve application performance, use the **spt_select_single_np( )** function instead of the **spt_select( )** function. For multiple file desciptors, use the **spt_select( )** function.

In **spthread.h**, a mapping of **select( )** to **spt_select_single_np( )** has been defined:

```
#define select(nfds, readfds, writefds, errorfds, timeout)\
        spt_select_single_np(nfds, readfds, writefds, errorfds, timeout)
```

For C applications that do not use the nonblocking feature, this mapping is available only when the correct two preprocessors have been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE
#define SPT_SELECT_SINGLE
#include <spthread.h>
```

For C applications that use the nonblocking feature, this mapping is available only when the correct two preprocessors have been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE_NONBLOCK
#define SPT_SELECT_SINGLE
#include <spthread.h>
```

For C++ applications that do not use the nonblocking feature, this mapping is available only when the correct two preprocessors have been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE_PRAGMA
#define SPT_SELECT_SINGLE
#include <spthread.h>
```

For C++ applications that use the nonblocking feature, this mapping is available only when the correct two preprocessors have been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK
#define SPT_SELECT_SINGLE
#include <spthread.h>
```

## NOTES

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

To use a combination of the **spt_select( )** and the **spt_select_single_np( )** functions in a single source file, you must explicitly call these functions.

## RETURN VALUES

See the **select(2)** reference page. The following information also applies:

- If the file descriptor becomes invalid (is closed by another thread), -1 is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## RELATED INFORMATION

Functions: **select(2)**, **spt_select(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_send -** Initiates thread-aware **send( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <spthread.h>**

**ssize_t spt_send(**
**int** *socket***,**
**const void ***buffer***,**
**size_t** *length***,**
**int** *flags***);**

## PARAMETERS

See the **send(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **send( )** function. The socket must be nonblocking for this function to be thread-aware.

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

The following macro maps **spt_send( )** to **send( )** and has been defined in **spthread.h**:

```
#define send(socket, buffer, length, flags)\
        spt_send(socket, buffer, length, flags)
```

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

For more description and notes, see the **send(2)** reference page.

## RETURN VALUES

See the **send(2)** reference page. The following also applies:

- The value of **errno** is never set to [EWOULDBLOCK].
- If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## ERRORS

See the **send(2)** reference page.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_sendmsg -** Initiates thread-aware **sendmsg( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <spthread.h>**

```
ssize_t spt_sendmsg(
        int socket,
        const struct msghdr *message,
        int flags);
```

## PARAMETERS

See the **sendmsg(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **sendmsg( )** function. The socket must be nonblocking for this function to be thread-aware.

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

The following macro maps **spt_sendmsg( )** to **sendmsg( )** and has been defined in **spthread.h**:

```
#define sendmsg(socket, message, flags)\
        spt_sendmsg(socket, message, flags)
```

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE
```

For more description and notes the **sendmsg(2)** reference page.

## RETURN VALUES

See the **sendmsg(2)** reference page. The following also applies:

- The value of **errno** is never set to [EWOULDBLOCK].
- If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## ERRORS

See the **sendmsg(2)** reference page.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_sendmsgx -** Sends a message on a socket using a message structure (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <sys/socket.h>**]
**#include <spthread.h>**

**ssize_t spt_sendmsgx(**
    **int** *socket***,**
    **const struct msghdr ****message***,**
    **int** *flags*
    **);**

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*message* Points to a **msghdr** structure containing both the destination address for the outgoing message and the buffers for the outgoing message. The length and format of the address depend on the address family for the socket. The **msg_flags** member of the structure is ignored. For:

**AF_INET** sockets
A pointer in **msghdr** to the address structure **sockaddr_in** must be cast as a **struct sockaddr**.

**AF_INET6** sockets
A pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**.

**AF_UNIX** sockets
A pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*flags* Is a value that controls message transmission. The value of the *flags* parameter is formed by bitwise ORing zero or more of these values:

**MSG_DONTROUTE**
Sends without using routing tables. (Not recommended; use only for debugging.)

**MSG_OOB** Sends out-of-band data on sockets that support out-of-band communications.

## DESCRIPTION

The **spt_sendmsgx( )** function is a thread-aware version of the **sendmsg( )** function.

The **spt_sendmsgx( )** function sends a message through a connection-oriented or connectionless socket. If the socket is connectionless, the message is sent to the address specified in the **msghdr** structure. If the socket is connection-oriented, the destination address in the **msghdr** structure is ignored.

Successful completion of a call to **spt_sendmsgx( )** does not imply successful delivery of the message. A return value of -1 indicates only locally detected errors.

If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **spt_sendmsgx( )** function blocks until space is available. If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **spt_sendmsgx( )** function fails and sets **errno** to [EWOULDBLOCK].

In the **msghdr** structure, the **msg_control** and **msg_controllen** members specify the ancillary data buffer that only sockets in the **AF_UNIX** domain can use to pass file descriptors to another process on the same node. The **msg_control** member can be a null pointer if ancillary data is not desired or required. If the **msg_control** member is nonnull, it points to an ancillary data buffer consisting of a **cmsghdr** structure followed by 1 to 16 file descriptors. The **msg_controllen** member specifies the size of the ancillary data buffer.

If **spt_sendmsgx( )** is called with an ancillary data buffer, the members of the **cmsghdr** structure must be set as follows:

- The **cmsg_level** member must be set to **SOL_SOCKET**.
- The **cmsg_type** member must be set to **SCM_RIGHTS**.
- The value of the **cmsg_len** member must be equal to the value of the **msg_controllen** member of the **msghdr** structure.

## NOTES

The macro to map **sendmsg( )** to **spt_sendmsgx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **sendmsg( )** to **spt_sendmsgx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

When data can be sent, a call to the **select( )** function indicates that the file descriptor for the socket is ready for writing.

## RETURN VALUES

Upon successful completion, the **spt_sendmsgx( )** function returns the number of normal bytes sent. Ancillary data, if present, is not counted in the total number of bytes sent.

If the **spt_sendmsgx( )** function call fails, the value -1 is returned, and **errno** is set to indicate the error.

If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

**ERRORS**

If any of these conditions occur, the **spt_sendmsgx( )** function sets **errno** to the corresponding value:

[EACCES] The socket is in the **AF_UNIX** domain and either search permission is denied for a component of the pathname in the **msghdr** structure or write access to the specified socket is denied.

[EAFNOSUPPORT] You cannot use addresses in the specified address family with this socket.

[EBADF] One of these conditions exists:

- The *socket* parameter is not a valid file descriptor.
- The socket is in the **AF_UNIX** domain, and one or more of the file descriptors being passed is invalid.

[ECONNRESET] One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EDESTADDRREQ] The socket is not connection-oriented, no peer address is set, and no destination address is specified.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EINTR] A signal interrupted the function before any data was transmitted.

[EINVAL] One of these conditions occurred:

- The socket is in the **AF_UNIX** domain, and the **msg_control** member contains either more than 16 file descriptors or fewer than 1 file descriptor.
- The socket is in the **AF_UNIX** domain, and an attempt was made to send more than one **cmsghdr** structure.
- The socket is in the **AF_UNIX** domain, and the value of the **cmsg_len** member is not equal to the value of the **msg_controllen** member.
- The socket is in the **AF_UNIX** domain, and the **cmsg_type** member is not equal to **SCM_RIGHTS**.
- The sum of the values specified for the **msg_iovlen** member of the **msghdr** structure is too large for a data item of type **ssize_t**.

[EIO] The socket is in the **AF_UNIX** domain, and the transport agent failed to inherit the file descriptors being passed, or an input or output error occurred.

[ELOOP] The socket is in the **AF_UNIX** domain, and too many symbolic links were encountered in translating the pathname specified by the **msghdr** structure.

[EMSGSIZE] The message is too large to be sent all at once, as required by the socket.

[ENAMETOOLONG]
The socket is in the **AF_UNIX** domain, and one of these conditions exists:

- The pathname in the **msghdr** structure exceeds **PATH_MAX** characters.
- A component of the pathname in the **msghdr** structure exceeds **NAME_MAX** characters.
- The intermediate result of pathname resolution when a symbolic link is part of the pathname in the **msghdr** structure exceeds **PATH_MAX** characters.

You can call the **pathconf( )** function to obtain the applicable limits.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOENT] The socket is in the **AF_UNIX** domain, and one of these conditions occurred:

- A component of the pathname in the **msghdr** structure does not name an existing file.
- The **msghdr** structure specifies an empty string as a pathname.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOPROTOOPT]
The socket is in the **AF_UNIX** domain, and the **cmsg_level** member is not equal to **SOL_SOCKET**.

[ENOTCONN] The socket is connection-oriented but is not connected.

[ENOTDIR] The socket is in the **AF_UNIX** domain, and the pathname specified by the **msghdr** structure contains a component that is not a directory.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[EPERM] The address included in the *message* parameter is bound to a socket whose mode is different than the mode of the socket specified by the *socket* parameter.

[EPIPE] One of these conditions occurred:

- An attempt was made to send a message on a socket that is shut down for writing.

- An attempt was made to send a message on a connection-oriented socket, and the peer socket is closed or shut down for reading. The **SIGPIPE** signal is also sent to the calling process.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set), and the operation would block.

## RELATED INFORMATION

Functions: **fcntl(2)**, **getsockopt(2)**, **recv(2)**, **recvfrom(2)**, **recvmsg(2)**, **select(2)**, **send(2)**, **sendto(2)**, **sendmsg(2)**, **setsockopt(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **socketpair(2)**, **spt_recvx(2)**, **spt_recfromx(2)**, **spt_recvmsgx(2)**, **spt_send(2)**, **spt_sendx(2)**, **spt_sendtox(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_sendto -** Initiates thread-aware **sendto( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#define _XOPEN_SOURCE_EXTENDED 1**
**#include <spthread.h>**

**ssize_t spt_sendto(**
    **int** *socket*,
    **const void** ***buffer*,
    **size_t** *length***,**
    **int** *flags***,**
    **const struct sockaddr** ***dest_addr*,**
    **size_t** *dest_len***);**

## PARAMETERS

See the **sendto(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **sendto( )** function. The socket must be nonblocking for this function to be thread-aware.

This function requires that the feature-test macro _XOPEN_SOURCE_EXTENDED be specified when you compile the module.

The following macro maps **spt_sendto( )** to **sendto( )** and has been defined in **spthread.h**:

```
#define sendto(socket, buffer, length, flags, dest_addr, dest_len) \
    spt_sendto(socket, buffer, length, flags, dest_addr, dest_len)
```

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE
```

For more description and notes, see the **sendto(2)** reference page.

## RETURN VALUES

See the **sendto(2)** reference page. The following information also applies:

- The value of **errno** is never set to [EWOULDBLOCK].
- If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## ERRORS

See the **sendto(2)** reference page.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_sendtox -** Sends a message on a socket (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <sys/socket.h>**]
**#include <spthread.h>**

**ssize_t spt_sendtox(**
  **int** *socket***,**
  **const void** ***message***,**
  **size_t** *length***,**
  **int** *flags***,**
  **const struct sockaddr** ***dest_addr***,**
  **size_t** *dest_len*
  **);**

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*message* Points to the buffer containing the message to be sent.

*length* Specifies the length in bytes of the message to be sent.

*flags* Is a value that controls message transmission. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_DONTROUTE**
Sends without using routing tables. (Not recommended; use for debugging only.)

**MSG_OOB** Sends out-of-band data on sockets that support out-of-band communications.

*dest_addr* Points to a **sockaddr** structure that contains the destination address. The length and format of the address depends on the address family of the socket. For:

**AF_INET** sockets
A pointer to the address structure **sockaddr_in** must be cast as a **struct sockaddr**.

**AF_INET6** sockets
A pointer to the address structure **sockaddr_in6** must be cast as a **struct sockaddr**.

**AF_UNIX** sockets
A pointer to the address structure **sockaddr_un** must be cast as a **struct sockaddr**.

*dest_len* Specifies the length of the **sockaddr** structure pointed to by the *dest_addr* parameter.

## DESCRIPTION

The **spt_sendtox( )** function is a thread-aware version of the **sendto( )** function.

The **spt_sendtox( )** function sends a message through a connection-oriented or connectionless socket. If the socket is connectionless, the message is sent to the address specified in the **sockaddr** structure pointed to by the *dest_addr* parameter. If the socket is connection-oriented, the *dest_addr* parameter is ignored.

Successful completion of a call to **spt_sendtox( )** does not imply successful delivery of the message. A return value of -1 indicates only locally detected errors.

If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **spt_sendtox( )** function blocks until space is available. If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **spt_sendtox( )** function fails and sets **errno** to [EWOULDBLOCK].

## NOTES

The macro to map **sendto( )** to **spt_sendtox( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **sendto( )** to **spt_sendtox( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

When data can be sent, a call to the **select( )** function indicates that the file descriptor for the socket is ready for writing.

## RETURN VALUES

Upon successful completion, the **spt_sendtox( )** function returns the number of bytes sent. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_sendtox( )** function sets **errno** to the corresponding value:

[EACCES] The socket is in the **AF_UNIX** domain and either search permission is denied for a component of the pathname in the **sockaddr** structure, or write access to the specified socket is denied.

[EAFNOSUPPORT]
You cannot use addresses in the specified address family with this socket.

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EDESTADDRREQ]
The socket is not connection-oriented and does not have its peer address set, and no destination address was specified.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EHOSTUNREACH]
The destination host cannot be reached.

[EINTR] A signal interrupted the function before any data was transmitted.

[EIO] The socket is in the **AF_UNIX** domain and an input or output error occurred.

[EINVAL] The *dest_len* parameter is not a valid length for the address family.

[ELOOP] The socket is in the **AF_UNIX** domain and too many symbolic links were encountered in translating the pathname in the **sockaddr** structure.

[EMSGSIZE] The message is too large to be sent all at once, as required by the socket.

[ENAMETOOLONG]
The socket is in the **AF_UNIX** domain and one of the following conditions exists:

- The pathname in the **sockaddr** structure exceeds **PATH_MAX** characters.
- A component of the pathname in the **sockaddr** structure exceeds **NAME_MAX** characters.
- The intermediate result of pathname resolution when a symbolic link is part of the pathname in the **sockaddr** structure exceeds **PATH_MAX** characters.

You can call the **pathconf( )** function to obtain the applicable limits.

[ENETDOWN]
The local interface used to reach the destination is down.

[ENETUNREACH]
No route to the network or host is present.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOENT] The socket is in the **AF_UNIX** domain and one of the following conditions exists:

- A component of the pathname specified in the **sockaddr** structure does not name an existing file.
- The **sockaddr** structure specifies an empty string as a pathname.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOTCONN] The socket is connection-oriented but is not connected.

[ENOTDIR] The socket is in the **AF_UNIX** domain and the pathname in the **sockaddr** structure contains a component that is not a directory.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[EPERM] The file name specified by the *dest_addr* parameter is bound to a socket whose mode is different than the mode of the socket specified by the *socket* parameter.

[EPIPE] One of the following conditions occurred:

- An attempt was made to send a message on a socket that is shut down for writing.
- An attempt was made to send a message on a connection-oriented socket, and the peer socket is closed or shut down for reading. The **SIGPIPE** signal is also sent to the calling process.

[EWOULDBLOCK]
The socket file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the operation would block.

## RELATED INFORMATION

Functions: **fcntl(2)**, **getsockopt(2)**, **recv(2)**, **recvfrom(2)**, **recvmsg(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **setsockopt(2)**, **shutdown(2)**, **sockatmark(2)**, **socket(2)**, **spt_recvx(2)**, **spt_recvfromx(2)**, **spt_recvmsgx(2)**, **spt_sendto(2)**, **spt_sendx(2)**, **spt_sendmsgx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_sendx -** Sends a message on a connected socket (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <sys/socket.h>**]
**#include <spthread.h>**

**ssize_t spt_sendx(**
  **int** *socket***,**
  **const void ***buffer***,**
  **size_t** *length***,**
  **int** *flags*
  **);**

## PARAMETERS

*socket* Specifies the file descriptor of the socket.

*buffer* Points to the buffer containing the message to send.

*length* Specifies the length in bytes of the message to send.

*flags* Is a value that controls message transmission. The value of the *flags* parameter is formed by bitwise ORing zero or more of the following values:

**MSG_DONTROUTE**
Sends without using routing tables. (Not recommended, use for debugging only.)

**MSG_OOB** Sends out-of-band data on sockets that support out-of-band communications.

## DESCRIPTION

The **spt_sendx( )** function is a thread-aware version of the **send( )** function.

The **spt_sendx( )** function begins transmission of a message to a peer socket. The **spt_sendx( )** function sends a message only when the socket is connected.

The length of the message to be sent is specified by the *length* parameter. If the message is too long to pass through the underlying protocol, the **spt_sendx( )** function fails and does not transmit the message.

Successful completion of a call to **spt_sendx( )** does not imply successful delivery of the message. A return value of -1 indicates only locally detected errors.

If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is blocking (**O_NONBLOCK** is not set), the **spt_sendx( )** function blocks until space is available. If the sending socket has no space to hold the message to be transmitted and the socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set), the **spt_sendx( )** function fails and sets **errno** to [EWOULDBLOCK].

## NOTES

The macro to map **send( )** to **spt_sendx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **send( )** to **spt_sendx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

When data can be sent, a call to the **select( )** function indicates that the file descriptor for the socket is ready for writing.

Calling the **spt_sendx( )** function with a *flags* parameter of 0 (zero) is identical to calling the **spt_writex( )** function.

## RETURN VALUES

Upon successful completion, the **spt_sendx( )** function returns the number of bytes sent. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

If the socket becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_sendx( )** function sets **errno** to the corresponding value:

[EBADF] The *socket* parameter is not a valid file descriptor.

[ECONNRESET]
One of the following conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The socket can only be closed.

[EDESTADDRREQ]
The socket is not connection-oriented and no peer address is set.

[EFAULT] A user-supplied memory buffer cannot be accessed.

[EINTR] A signal interrupted the function before any data was transmitted.

[EIO] An input or output error occurred.

[EMSGSIZE] The message is too large to be sent all at once, as required by the socket.

[ENETDOWN]
The local interface used to reach the destination is down.

[ENETUNREACH]
No route to the network or host is present.

[ENOBUFS] Not enough buffer space was available to complete the call. A retry at a later time might succeed.

[ENOMEM] Required memory resources were not available. A retry at a later time might succeed.

[ENOTCONN] The socket either is not connected or has not had the peer socket previously specified.

[ENOTSOCK] The *socket* parameter does not refer to a socket.

[EOPNOTSUPP]
The specified value for the *flags* parameter is not supported for this socket type or protocol.

[EPIPE] One of the following conditions occurred:

- An attempt was made to send a message on a socket that is shut down for writing.
- An attempt was made to send a message on a connection-oriented socket and the peer socket is closed or shut down for reading. The **SIGPIPE** signal is also sent to the calling process.

[EWOULDBLOCK]
The socket's file descriptor is marked nonblocking (**O_NONBLOCK** is set) and the operation would block.

## RELATED INFORMATION

Functions: **connect(2)**, **fcntl(2)**, **getsockopt(2)**, **recv(2)**, **recvfrom(2)**, **recvmsg(2)**, **select(2)**, **send(2)**, **sendmsg(2)**, **sendto(2)**, **setsockopt(2)**, **sockatmark(2)**, **shutdown(2)**, **socket(2)**, **spt_recvx(2)**, **spt_recvfromx(2)**, **spt_recvmsgx(2)**, **spt_send(2)**, **spt_sendtox(2)**, **spt_sendmsgx(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

**NAME**

**SPT_SETMODE -** Sets device-dependent Guardian file-system functions

**LIBRARY**

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

**SYNOPSIS**

```
[#include <cextdecs.h>]
#include <spthread.h>

short SPT_ SETMODE (
        short filenum,
        short function,
        [short param1 ],
        [short param2 ],
        [short *last_params]
        );
```

**PARAMETERS**

*filenum* Specifies the Guardian file number of a Guardian file open instance, identifying the file to receive the requested function.

*function* Specifies the number of a device-dependent function. For a description of valid values, see the table of SETMODE functions in the *Guardian Procedure Calls Reference Manual*.

*param1* Provides the first value or pattern of set bits that defines the specific function setting to be used. For a description of valid values, see the table of SETMODE functions in the *Guardian Procedure Calls Reference Manual*.

*param2* Provides the second value or pattern of set bits that defines the specific function setting to be used. For a description of valid values, see the table of SETMODE functions in the *Guardian Procedure Calls Reference Manual*.

*last_params* Returns the previous settings of *param1* and *param2* associated with the current function.

**DESCRIPTION**

The **SPT_SETMODE( )** function is the thread-aware version of the Guardian SETMODE procedure.

The **SPT_SETMODE( )** function is used to set device-dependent Guardian file-system functions. A call to the **SPT_SETMODE( )** function is rejected with an error indication if incomplete nowait operations are pending on the specified file.

For programming information about the Guardian SETMODE file-system procedure, see the *Guardian Programmer's Guide* and the manual for the data communication protocol you are using.

## Considerations

Default settings

The **SPT_SETMODE( )** settings designated as default in the *Guardian Procedure Calls Reference Manual* are the values that apply when a file is opened (not if a particular *function* value is omitted when **SPT_SETMODE( )** is called).

Waited **SPT_SETMODE( )** use

The **SPT_SETMODE( )** function is used on a file as a waited operation even if *filenum* has been opened for nowait operations. Use the Guardian SETMODENOWAIT procedure for nowait operations.

Use for Telserv processes

No **SPT_SETMODE( )** calls on Telserv are allowed before doing an **SPT_CONTROL( )** function 11.

Ownership and security of a disk file

"Set disk file security" and "set disk file owner" are rejected unless the requester is the owner of the file or the super ID.

### Interprocess Communication Considerations

Nonstandard parameter values

You can specify any value for the *function*, *param1*, and *param2* parameters. Establish an application-defined protocol for interpreting nonstandard parameter values.

User-defined functions

Use of *function* code numbers 100 to 109 avoids any potential conflict with **SPT_SETMODE( )** function codes defined by HP.

Incorrect use of *last_params*

Guardian file-system error 2 is returned when the *last_params* parameter is supplied but the target process does not correctly return values for this parameter.

Process message

Issuing an **SPT_SETMODE( )** call to a file representing another process causes a system message -33 (process SETMODE) to be sent to that process.

You can identify the process that called **SPT_SETMODE( )** in a subsequent call to the Guardian FILE_GETRECEIVEINFO_ (or LASTRECEIVE or RECEIVEINFO) procedure. For a list of all system messages sent to processes, see the *Guardian Procedure Errors and Messages Manual*.

## RETURN VALUES

The **SPT_SETMODE( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_setOSSFileIOHandler -** Sets interest in file descriptor

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**extern spt_error_t spt_setOSSFileIOHandler(**
**const int** *filedes***,**
**const int** *read***,**
**const int** *write***,**
**const int** *error***);**

## PARAMETERS

*filedes* Specifies the OSS file descriptor for the file of interest

*read* Nonzero indicates interest in read ready

*write* Nonzero indicates interest in write ready

*error* Nonzero indicates interest in exception pending

## DESCRIPTION

This function sets interest in an OSS file descriptor.

## RETURN VALUES

**SPT_SUCCESS**

This value is returned for any of the following conditions:

- The *filedes* interest was successfully set
- The *filedes* was not registered prior to this call
- The specified *filedes* is invalid
- The specified *filedes* is not supported

**SPT_ERROR** The specified *filedes* was less than 0 (zero).

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_setTMFConcurrentTransactions -** Sets the number of concurrent TMF transactions

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_setTMFConcurrentTransactions (**
**short** *max_trans* **);**

## PARAMETERS

*max_trans* Specifies the maximum number of concurrent transactions desired.

## DESCRIPTION

This function sets the maximum number of concurrent TMF transactions.

## RETURN VALUES

This function returns 0 (zero) upon successful completion of the call. If an error occurs, this function can return the following value:

**EINVAL** Unable to change the maximum number of concurrent transactions because TMF is already processing transactions.

## RELATED INFORMATION

Functions: **spt_getTMFConcurrentTransactions(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_sigaction -** Specifies the action to take upon delivery of a signal (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_sigaction(**
  **int** *sig***,**
  **const sigaction_t ***act***,**
  **sigaction_t ***o_action***);**

## PARAMETERS

*sig* Specifies the signal number.

*act* Points to a **sigaction_t** structure that describes the new action to be taken on delivery the signal identified by the *sig* parameter.

*o_action* Points to a **sigaction_t** structure that returns the signal action previously associated with the signal.

## DESCRIPTION

The **spt_sigaction( )** function allows the calling thread to change or examine the action to be taken on delivery of a specific signal. This call removes any previously established signal handler for this signal for this thread. You must reestablish the previous signal handler if you want to use it at a later time.

To catch externally generated signals (such as SIGINT, SIGQUIT, SIGALRM, and SIGCHLD) at the thread level, you must export the SPT_THREAD_AWARE_SIGNAL environmental variable to the value 1. By default, SPT_THREAD_AWARE_SIGNAL is disabled. If you export SPT_THREAD_AWARE_SIGNAL to 1, the signal handler registered for that thread will be executed immediately within the scope of the internal generic handler for pthreads. However, the thread itself executes only when it is scheduled. Consequently, you cannot use thread-specific functions like **pthread_self( )** inside thread-specific signal handler functions.

You should not use the **SA_ONSTACK** flag and the **SA_ONSTACK_COMPATIBILITY** feature test macro in a threaded application that uses the Standard POSIX Threads library. Use of these two options can result in undefined behavior in the SPT environment.

Every signal has an associated default action. The **spt_signal( )** function can change this action by specifying that the receiving thread:

- Ignore the delivery of a specific signal.
- Restore the default action for a specific signal.
- Invoke a signal-catching function in response to the delivery of a specific signal.

For the defined signal names and details about the cause and default action of each defined signal, see the **signal(4)** reference page

## NOTES

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

Upon successful completion, the **spt_sigaction( )** function returns the value 0 (zero). Otherwise, the value -1 is returned, no new signal handler is installed, and **errno** is set to indicate the error.

**ERRORS**

If one of these conditions occurs, the **spt_sigaction( )** function sets **errno** to [EINVAL]:

[EINVAL] One of the following conditions exists:

- The *sig* parameter is not a valid signal number.
- An attempt was made to ignore or supply a signal-catching function for the **SIGKILL**, **SIGSTOP**, or **SIGABEND** signal.
- The signal is not supported in pthreads.

**RELATED INFORMATION**

Functions: **pthread_kill(2)**, **pthread_sigmask(2)**, **sigaction(2)**, **spt_pause(2)**, **spt_signal(2)**, **spt_sigsuspend(2)**.

**STANDARDS CONFORMANCE**

The **spthread.h** header file is an HP extension and an HP exception to the IEEE Std 1003.1c-1995, *POSIX System Application Program Interface*.

## NAME

**spt_signal -** Installs a new signal handler

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**void *spt_signal(int** *sig***, void (*** *handler***)(int))**

## PARAMETERS

*sig* Specifies the signal number.

*handler* User-specified signal function that acts a signal handler.

## DESCRIPTION

The **spt_signal( )** function allows the calling thread to change the action to be taken when a specific signal is delivered to the thread calling this function.

To catch externally generated signals (such as SIGINT, SIGQUIT, SIGALRM, and SIGCHLD) at the thread level, you must export the SPT_THREAD_AWARE_SIGNAL environmental variable to the value 1. By default, SPT_THREAD_AWARE_SIGNAL is disabled. If you export SPT_THREAD_AWARE_SIGNAL to 1, the signal handler registered for that thread will be executed immediately within the scope of the internal generic handler for pthreads. However, the thread itself executes only when it is scheduled. Consequently, you cannot use thread-specific functions like **pthread_self( )** inside thread-specific signal handler functions.

Every signal has an associated default action. The **spt_signal( )** function can change this action by specifying that the receiving thread:

- Ignore the delivery of a specific signal.
- Restore the default action for a specific signal.
- Invoke a signal-catching function in response to the delivery of a specific signal.

If you use the **spt_signal( )** function to invoke a signal-catching function, the action associated with the signal is restored to the default action each time the signal is delivered. This behavior is different from the behavior of the **spt_sigaction( )** function, which does not restore the default signal action after execution.

In **spthread.h**, a mapping of **signal( )** to **spt_signal( )** has been defined:

**#define signal(***sig_handler***) spt_signal((***sig***),(handler**))

For C applications, this mapping is available only when you define the correct preprocessor before you include **spthread.h**:

**#define SPT_THREAD_SIGNAL**
**#include <spthread.h>**

For C++ applications, this mapping is available only when you define the correct preprocessor before you include **spthread.h**:

**#define SPT_THREAD_SIGNAL_PRAGMA**
**#include <spthread.h>**

**NOTES**

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

Upon successful completion the **spt_signal( )** function returns the value of the previous signal action. Otherwise, the value SIG_ERR is retuned and **errno** is set indicate the error.

NOTE: SIG_ERR is not a valid value for a signal-catching function.

**ERRORS**

If any of these conditions occur, **errno** is set to the corresponding value:

[EINVAL] The value of the **sig** argument is not a valid signal number, or an attempt was made to ignore or supply a signal-catching function for the SIGKILL, SIGSTOP, or SIGABEND signal.

**RELATED INFORMATION**

Functions: **pthread_kill(2)**, **pthread_sigmask(2)**, **signal(3)**, **spt_pause(2)**, **spt_sigaction(2)**, **spt_sigsuspend(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_sigpending -** Examines signals that are blocked and pending

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_sigpending(sigset_t ****set***);**

## PARAMETERS

*set* Specifies the set of signals that are blocked and pending.

## DESCRIPTION

The **spt_sigpending(2)** function retrieves the signals that have been sent to the calling thread but have been blocked from delivery. These signals are pending to the calling thread, the calling thread's signal mask is preventing their delivery. The blocked signals are stored in the structure pointed to by the *set* parameter. Because signals can arrive asynchronously, do not make assumptions about the current set of pending signals based on the value returned by this function in *set*.

## NOTES

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

Upon successful completion, the **spt_sigpending(2)** function returns a value of 0 (zero). Otherwise -1 is returned and **errno** is set to indicate the error.

## ERRORS

If this conditions occurs, the **spt_sigpending(2)** function sets **errno** to the corresponding value:

[EFAULT] The *set* argument points to an invalid address.

## RELATED INFORMATION

Functions: **pthread_kill(2)**, **pthread_sigmask(2)**, **sigpending(2)**, **spt_pause(2)**, **spt_sigaction(2)**, **spt_signal(2)**, **spt_sigsuspend(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_sigsuspend -** Changes the set of blocked signals and waits for a signal

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_sigsuspend(const sigset_t ***sset***);**

## PARAMETERS

*sset* Specifies the set of signals to be blocked from delivery to the calling thread.

## DESCRIPTION

The **spt_sigsuspend( )** function replaces the current signal mask of a thread with the signal set specified by the *sset* parameter and suspends processing for the thread until the thread receives one of the following signals:

- SIGSTOP, SIGKILL, or SIGABEND.
- A signal that is not a member of *sset* and has an action that:
  - — Calls a signal-catching function.
  - — Ends the request.
  - — Terminates the process.

NOTE: The signal mask specifies a set of signals to be blocked. A function does not receive or respond to signals that are specified in the signal mask. However, the signals SIGSTOP, SIGKILL, and SIGABEND cannot be blocked or ignored, even if they are specified in the signal mask.

If an incoming unblocked signal has an action to terminate, the **spt_sigsuspend( )** function never returns a value. If a signal-catching function handles an incoming signal, the **spt_sigsuspend( )** function returns only after the signal-catching function returns. In this case, the signal mask of the thread is restored to whatever it was before the **spt_sigsuspend( )** function was called.

To catch externally generated signals (such as SIGINT, SIGQUIT, SIGALRM, and SIGCHLD) at the thread level, you must export the SPT_THREAD_AWARE_SIGNAL environmental variable to the value 1. By default, SPT_THREAD_AWARE_SIGNAL is disabled.

## NOTES

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

### RETURN VALUES

When the signal is caught by the calling thread and control is returned from the signal-catching function, the calling thread resumes execution from the point of suspension. The **spt_sigsuspend(2)** function always returns -1 and sets **errno** to the value [EINTR].

### ERRORS

The **spt_sigsuspend(2)** function alway sets **errno** to the following value:

[EINTR] The signal is caught by the calling thread and control is returned from the signal-catching function.

### RELATED INFORMATION

Functions: **pthread_kill(2)**, **pthread_sigmask(2)**, **sigsuspend(2)**, **spt_pause(2)**, **spt_sigaction(2)**, **spt_signal(2)**.

### STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_sigwait -** Causes the calling thread to wait for a signal

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_sigwait(**
  **sigset_t ***set*
  **int ***sig***);**

## PARAMETERS

*set* Specifies the set of signals that the calling thread will wait for.

*sig* Receives the signal number cleared from the specified set of signal numbers.

## DESCRIPTION

This function causes a thread to wait for a signal. It atomically chooses a pending signal from the set of pending signals indicated by the *set* parameter, atomically clears that signal from the system's set of pending signals, and returns that signal number at the location specified by the *sig* parameter. If no signal in *set* is pending at the time of the call, the thread is blocked until one or more signals become pending. The signals defined by *set* should be unblocked during the call to this function and are blocked when the thread returns from the call, unless another thread is currently waiting for one of those signals.

A thread must block the signals it waits for using the **pthread_sigmask( )** function before calling this function.

If more than one thread is using this function to wait for the same signal, only one of those threads returns from this function with the signal number.

A call to the **spt_sigwait( )** function is a cancellation point.

To catch externally generated signals (such as SIGINT, SIGQUIT, SIGALRM, and SIGCHLD) at the thread level, you must export the SPT_THREAD_AWARE_SIGNAL environmental variable to the value 1. By default, SPT_THREAD_AWARE_SIGNAL is disabled.

## NOTES

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

HP recommends that you do not specify threads to wait for process-level signals like SIGCONT, SGTTIN, SIGTTOU, and SIGSTP. If a thread uses a function like **spt_sigwait( )** or **spt_sigsuspend( )**, the thread breaks from the wait state only if the corresponding signal is sent using the **pthread_kill( )** function. The thread does not break from the wait state for signals that are generated externally at the process level.

The SIGCHLD signal is delivered to the correct thread even though the SIGCHILD signal is generated asynchronously.

**RETURN VALUES**

On a successful call, the signal number is returned. Otherwise the error [EINVAL] is returned.

**ERRORS**

If the only signals passed are unsupported signals, the **spt_sigwait( )** function returns the error [EINVAL]. For some signals, support by the **spt_sigwait( )** function depends on the RVU running on the system:

- The SIGUNCP signal is supported on systems running H-series RVUs only.
- For H06.06 and later H-series RVUs, and G06.29 and later G-series RVUs only, these signals are supported:
  - SIGCONT
  - SIGTTIN
  - SIGTTOU
  - SIGCHLD
  - SIGTSTP

**RELATED INFORMATION**

Functions: **pause(2)**, **pthread_cancel(2)**, **pthread_sigmask(2)**, **sigpending(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 Version 2 specification.

Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The HP implementation does not provide the **spt_sigwaitinfo( )** or **sigtimedwait( )** functions.

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_sleep -** Suspends execution of the thread for a specified time interval

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**unsigned int spt_sleep(**
**unsigned int** *seconds***);**

## PARAMETERS

*seconds* Specifies the number of seconds for which the thread is to be suspended.

## DESCRIPTION

The **spt_sleep( )** function suspends a thread for a specified number of seconds. A certain amount of delay can be expected in the processing of the **spt_sleep( )** call because of other processor-intensive or input/output-intensive threads. If an unblocked signal is received during the suspension period, **spt_sleep( )** returns control immediately and returns the sleep time remaining.

## NOTES

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll*nnn*/zsptdll**).

## RETURN VALUES

This function can return the following values:

**0** (zero) The thread was suspended for the full time specified.

*seconds* Indicates the number of seconds remaining in the specified suspension time.

## RELATED INFORMATION

Functions: **spt_usleep(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_system -** Initiates thread-aware **system( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_system(**
**const char** ***command*);**

## PARAMETERS

See the **system(3)** reference page.

## DESCRIPTION

This is a thread-aware version of the **system( )** function. All threads in the process are temporarily blocked while the child process, which performs the **system( )** call, is created.

In **spthread.h**, a mapping of **system( )** to **spt_system( )** has been defined:

**#define system(command) spt_system(command)**

For C applications that do not use the nonblocking feature, this mapping is available only when the correct preprocessor has been defined before including **spthread.h** as follows:

**#define SPT_THREAD_AWARE**
**#include <spthread.h>**

For C applications that use the nonblocking feature, this mapping is available only when the correct preprocessor has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE_NONBLOCK**
**#include <spthread.h>**

For C++ applications that do not use the nonblocking feature, this mapping is available only when the correct preprocessor has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE_PRAGMA**
**#include <spthread.h>**

For C++ applications that use the nonblocking feature, this mapping is available only when the correct preprocessor has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**
**#include <spthread.h>**

## RETURN VALUES

See the **system(3)** reference page. Also, if a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_TimerHandler_p -** Executes callback type required by **spt_regTimerHandler( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <spthread.h>

typedef long (*spt_TimerHandler_p)(void);
```

## PARAMETERS

None.

## DESCRIPTION

This function executes a callback type required by the **spt_regTimerHandler( )** function. The callback is executed in the context of the last running thread. This means that the callback executes on the stack of the last running thread.

## RETURN VALUES

| | |
|---|---|
| 0 | Callback has readied a thread to run, and will be invoked again as soon as possible. |
| -1 | Callback has not readied a thread, but will be invoked again as soon as possible. |
| >0 (zero) | Callback has not readied a thread. Return value is the hundredths of a second until callback should be invoked again. |

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_TMF_GetTxHandle -** Gets the current TMF transaction handle

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**short SPT_TMF_GetTxHandle(**
**SPT_TMF_TxHandle_t ****tx_handle* **);**

## PARAMETERS

*tx_handle* Receives the current active TMF transaction handle.

## DESCRIPTION

This function retrieves the current active transaction handle of the thread.

## RETURN VALUES

This function returns an integer value indicating the result of the call. Possible return values are:

**0** (zero) Successful completion of the call. The current active transaction handle is returned in *tx_handle*.

**22** A bounds error occurred.

**29** There are missing parameters.

**75** There is no current transaction.

## RELATED INFORMATION

Functions: **SPT_TMF_SetTxHandle(2)**, **SPT_TMF_Init(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_TMF_Init -** Initializes the tfile for concurrent transaction management

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**short SPT_TMF_Init( void );**

## PARAMETERS

None.

## DESCRIPTION

This function opens the tfile for concurrent transaction management.

## RETURN VALUES

**SPT_SUCCESS**
The TMF file is initialized for concurrent transaction management.

*error* Contains the error value returned by the underlying call to the Guardian OPEN procedure. See the *Guardian Procedure Errors and Messages Manual* for more information on the specific value returned.

## RELATED INFORMATION

Functions: **SPT_TMF_GetTxHandle(2)**, **SPT_TMF_SetTxHandle(2)**, **spt_getTMFConcurrentTransactions(2)**, **spt_setTMFConcurrentTransactions(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX98 specification. Interfaces documented on this reference page conform to the IEEE Std 1003.1c-1995, POSIX System Application Program Interface.

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_TMF_RESUME -** Resumes a previously suspended transaction associated with the current thread

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series and J-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**short SPT_TMF_RESUME(**
**long long** **txid*
**);**

## PARAMETERS

### Input

*txid* Specifies the transactional identifier returned by **SPT_TMF_SUSPEND( )** or TMF_GET_TX_ID.

## DESCRIPTION

This function resumes a previously suspended transaction associated with the current thread.

## RETURN VALUES

A status word is returned. The value is one of the following:

0 (zero) The **SPT_TMF_RESUME( )** operation completed successfully.

Nonzero values
The Guardian file-system error with this error number occurred.

## RELATED INFORMATION

Functions: **SPT_TMF_SUSPEND(2)**.

## NAME

**SPT_TMF_SetAndValidateTxHandle** - Sets the current TMF transaction handle to be associated with the current thread

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**short SPT_TMF_SetAndValidateTxHandle(**
**SPT_TMF_TxHandle_t ****tx_handle***);**

## PARAMETERS

*tx_handle* Specifies the transaction handle of the current TMF transaction.

## DESCRIPTION

This function sets the specified transaction handle as the current active transaction for the thread. In addition, it validates the transaction. If the transaction is not valid, the transaction is aborted.

## RETURN VALUES

This function returns an integer value indicating the result of the call. Possible return values are:

0 (zero) The **SPT_TMF_SetAndValidateTxHandle( )** operation completed successfully; the transaction handle was successfully set and validated.

Nonzero values
The Guardian file-system error with this error number occurred.

## NAME

**SPT_TMF_SetTxHandle -** Sets the TMF transaction handle

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**short SPT_TMF_SetTxHandle(**
**SPT_TMF_TxHandle_t ****tx_handle* **);**

## PARAMETERS

*tx_handle* Specifies the transaction handle of the current TMF transaction.

## DESCRIPTION

This function sets the specified transaction handle as the current active transaction for the thread.

## RETURN VALUES

This function returns an integer value indicating the result of the call. Possible return values are:

**0** (zero) Indicates the transaction handle was successfully set.

**22** Indicates that a bounds error occurred.

**29** Indicates missing parameters.

**75** Indicates that there is no current transaction.

**78** Indicates an invalid transaction identifier or that a transaction has not started on this Expand node.

**715** Indicates an invalid transaction handle.

## RELATED INFORMATION

Functions: **SPT_TMF_GetTxHandle(2)**, **SPT_TMF_Init(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file i**spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_TMF_SUSPEND -** Suspends a transaction associated with the current thread

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**short SPT_TMF_SUSPEND(**
  **long long** **txid*
 **);**

## PARAMETERS

### Output

*txid* Returns a transactional identifier that can be used for a subsequent **SPT_TMF_RESUME( )** call.

## DESCRIPTION

This function suspends a transaction associated with the current thread.

## RETURN VALUES

A status word is returned. The value is one of the following:

0 (zero) The **SPT_TMF_SUSPEND( )** operation completed successfully.

Nonzero values
The Guardian file-system error with this error number occurred.

## RELATED INFORMATION

Functions: **SPT_TMF_RESUME(3)**.

**NAME**

**SPT_UNLOCKFILE -** Unlocks a disk file and any records in that file currently locked by the user

**LIBRARY**

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

**SYNOPSIS**

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_UNLOCKFILE (**
**short** *filenum***,**
[**long** *tag* ]
**);**

**PARAMETERS**

*filenum* specifies the Guardian file number of a Guardian file open instance for the file that you want unlocked.

*tag* is for nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

**DESCRIPTION**

The **SPT_UNLOCKFILE()** function is the thread-aware version of the Guardian UNLOCKFILE procedure.

The **SPT_UNLOCKFILE()** function unlocks a disk file and any records in that file currently locked by the user. The user is defined either as the opener of the file (identified by the *filenum* value used) if the file is not audited, or by the transaction (identified by the TRANSID) if the file is audited. Unlocking a file allows other processes to access the file. This call has no affect on an audited file if the current transaction has modified that file.

For programming information about the Guardian UNLOCKFILE file-system procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

**Considerations**

Nowait and **SPT_UNLOCKFILE()**
The **SPT_UNLOCKFILE()** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

Locking queue If any users are queued in the locking queue for the file, the process at the head of the locking queue is granted access and is removed from the queue (the next read or lock request moves to the head of the queue). If the next user in the locking queue is waiting to:

- lock the file or lock a record in the file, the user is granted the lock (which excludes other users from accessing the file) and resumes processing.

- read the file, its read is processed.

Transaction Management Facility (TMF) and **SPT_UNLOCKFILE( )**
If the current transaction modifies a file audited by TMF, locks on the file are released only when TMF ends or aborts the transaction. In other words, a locked audited file that the current transaction modified is unlocked during **SPT_ENDTRANSACTION( )** or **SPT_ABORTTRANSACTION( )** processing for that file. You can use the **SPT_UNLOCKFILE( )** function to unlock an unmodified audited record.

### Use on OSS Objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **SPT_UNLOCKFILE( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_UNLOCKREC -** Unlocks a Guardian file record currently locked by the user

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_UNLOCKREC (**
**short** *filenum***,**
[**long** *tag* ]
**);**

## PARAMETERS

*filenum* specifies the Guardian file number of a Guardian file open instance for the file containing the record you want unlocked.

*tag* is for nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **SPT_UNLOCKREC()** function is the thread-aware version of the Guardian UNLOCKREC procedure.

The **SPT_UNLOCKREC()** function unlocks a record in the specified file currently locked by the user. The user is defined either as the opener of the file (identified by the *filenum* value used) if the file is not audited, or by the transaction (identified by the TRANSID) if the file is audited.

This call unlocks the record at the current position in the file, allowing other users to access that record. This call has no affect on a record of an audited file if the current transaction has modified that record.

For programming information about the Guardian UNLOCKREC file-system procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

### Considerations

File opened nowait and **SPT_UNLOCKREC()**
The **SPT_UNLOCKREC()** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

Locking queue If any users are queued in the locking queue for the record, the user at the head of the locking queue is granted access and is removed from the queue (the next read or lock request moves to the head of the queue).

If the user granted access is waiting to lock the record, the user is granted the lock (which excludes other process from accessing the record) and resumes processing. If the user granted access is waiting to read the record, its read is processed.

Calling **SPT_UNLOCKREC( )** after KEYPOSITION

If the call to **SPT_UNLOCKREC( )** immediately follows a call to KEYPOSITION where a nonunique alternate key is specified, the **SPT_UNLOCKREC( )** call fails. A subsequent call to FILE_GETINFO_ or FILEINFO shows that Guardian file-system error 46 (`invalid key`) occurred. However, if an intermediate call to **SPT_READX( )** or **SPT_READLOCKX( )** is performed, the call to **SPT_UNLOCKREC( )** is permitted.

Unlocking several records

If several records need to be unlocked, you can call the **SPT_UNLOCKREC( )** function to unlock all records currently locked by the user (rather than unlocking the records through individual calls to **SPT_UNLOCKREC( )**).

Current-state indicators after **SPT_UNLOCKREC( )**

For key-sequenced, relative, and entry-sequenced files, the current-state indicators after an UNLOCKREC remain unchanged.

File pointers after **SPT_UNLOCKREC( )**

For unstructured files, the current-record pointer and the next-record pointer remain unchanged.

Transaction Management Facility (TMF) and **SPT_UNLOCKREC( )**

If the current transaction modifies a record in file audited by TMF, locks on the record are released only when TMF ends or aborts the transaction. In other words, a locked record in an audited file that the current transaction modified is unlocked during **SPT_ENDTRANSACTION( )** or **SPT_ABORTTRANSACTION( )** processing for that file. You can use the **SPT_UNLOCKREC( )** function to unlock an unmodified audited record.

### Use on OSS Objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **SPT_UNLOCKREC( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_unregFile -** Unregisters a Guardian file number as one that the user manages

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**extern spt_error_t spt_unregFile(**
**const short** *filenum***);**

## PARAMETERS

*filenum* Specifies the Guardian file number being unregistered

## DESCRIPTION

This function unregisters a Guardian file number as one that the user manages. Any threads waiting on file number I/O will awaken with **SPT_ERROR** and Guardian file-system error 16.

## RETURN VALUES

**SPT_SUCCESS**
The specified *filenum* was successfully unregistered.

**SPT_ERROR** One of the following conditions exists:

The value specified for *filenum* s less than 0 (zero)

- The specified *filenum* was not registered prior to this call
- The FILE_COMPLETE_SET_ procedure removal of *filenum* returned a nonzero value.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_unregOSSFileIOHandler -** Unregisters an OSS file descriptor

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**extern spt_error_t spt_unregOSSFileIOHandler(**
**const int** *filedes***);**

## PARAMETERS

*filedes* Specifies the OSS file descriptor being unregistered

## DESCRIPTION

This function unregisters an OSS file descriptor as one that the user manages.

## RETURN VALUES

**SPT_SUCCESS**
The specified *filedes* was successfully unregistered.

**SPT_ERROR** The specified *filedes* is less than 0 (zero) or was not registered prior to this call.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_unregPathsendTagHandler -** Unregisters the user-supplied Pathsend tag

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**spt_error_t spt_unregPathsendTagHandler (**
**const long** *tag* **);**

## PARAMETERS

*tag* Specifies the Pathsend tag to be unregistered.

## DESCRIPTION

This function unregisters the specified Pathsend tag as a tag that user manages.

## RETURN VALUES

**SPT_SUCCESS**
The specified tag was unregistered.

**SPT_ERROR** The specified tag was never registered.

## RELATED INFORMATION

Functions: **spt_regPathsendTagHandler(2)**, **SPT_SERVERCLASS_DIALOG_ABORT_(2)**, **SPT_SERVERCLASS_DIALOG_BEGIN_(2)**, **SPT_SERVERCLASS_DIALOG_END_(2)**, **SPT_SERVERCLASS_DIALOG_SEND_(2)**, **SPT_SERVERCLASS_SEND_INFO_(2)**, **SPT_SERVERCLASS_SEND_(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_usleep -** Suspends execution of the thread for a specified number of microseconds

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_usleep(**
**unsigned int** *useconds***);**

## PARAMETERS

*useconds* Specifies the number of microseconds for which the thread is to be suspended. The value specified must be less than or equal to 1000000.

## DESCRIPTION

The **spt_usleep( )** function suspends a thread for a specified number of microseconds. A certain amount of delay can be expected in the processing of the **spt_usleep( )** call because of other processor-intensive or input/output-intensive threads.

## NOTES

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

The **spt_usleep( )** function returns the value 0 (zero) when the call completes successfully. Otherwise, **spt_usleep( )** returns -1 and sets **errno**.

## ERRORS

If the following condition occurs, **spt_usleep( )** sets **errno** to the corresponding value:

[EINTR] A **pthread_kill( )** function call received a signal that is not blocked, ignored, or handled.

[EINVAL] The value specified for the *useconds* parameter was greater than 1000000.

## RELATED INFORMATION

Functions: **spt_sleep(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_vfprintf -** Initiates thread-aware **vfprintf( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

```
int spt_vfprintf(
        FILE *stream,
        const char *format,
        va_list printarg);
```

## PARAMETERS

See the **vfprintf(3)** reference page either online or in the *Open System Services Library Calls Reference Manual*.

## DESCRIPTION

This is a thread-aware version of the **vfprintf( )** function. The file descriptor underlying the stream must be nonblocking for this function to be thread-aware.

The following macro maps **spt_vfprintf( )** to **vfprintf( )** and has been defined in **spthread.h**:

```
#define vfprintf(stream, format, printarg) \
        spt_vfprintf((stream), (format), (printarg))
```

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE
```

## RETURN VALUES

See **vfprintf(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying stream becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF].
- If a signal is received via he *Lpthread_kill( ) function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_vfprintfx -** Formats a variable number of parameters for output (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <stdarg.h>**]
[**#include <stdio.h>**]
**#include <spthread.h>**

**int spt_vfprintfx (**
**FILE ****stream***,**
**const char ****format***,**
**va_list** *printarg*
**);**

## PARAMETERS

*stream* Specifies the output stream.

*format* Specifies a character string that contains two types of objects:

- Plain characters, which are copied to the output stream.
- Conversion specifications, each of which causes zero or more items to be fetched from the **stdarg** parameter lists.

*printarg* Specifies the parameters to be printed.

## DESCRIPTION

The **spt_vfprintfx( )** function is the thread-aware version of the **vfprintf( )** function.

The **spt_vfprintfx( )** function formats and writes **stdarg** parameter lists.

This function is the same as the **spt_fprintfx( )** function, except that it is not called with a variable number of parameters. Instead, it is called with a parameter list pointer as defined by **stdarg**.

## NOTES

The macro to map **vfprintf( )** to **spt_vfprintfx( )** is available in C applications when **SPT_THREAD_AWARE_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_NONBLOCK**

The alias to link **vfprintf( )** to **spt_vfprintfx( )** is available in C++ applications when **SPT_THREAD_AWARE_PRAGMA_NONBLOCK** has been defined in the following manner before including **spthread.h**:

**#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK**

## EXAMPLES

The following example demonstrates how you can use the **spt_vfprintfx( )** function to write an error routine:

```
#include <stdarg.h>
#include <stdio.h>
#define SPT_THREAD_AWARE_NONBLOCK
#include <spthread.h>

void    error(char *funct, char *fmt, ...)
```

```
{
        va_list args;
        /*
        ** Display the name of the function that called error
        */
        spt_fprintfx(stderr, "ERROR in %s: ", funct);
        /*
        ** Display the remainder of the message
        */
        va_start(args, fmt);
        spt_vfprintfx(stderr, fmt, args);
        va_end(args);
        abort();
}
```

## RETURN VALUES

Upon successful completion, this function returns the number of bytes in the output string. Otherwise, a negative value is returned.

If the file descriptor underlying *stream* becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill()** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## ERRORS

The **spt_vfprintfx()** function fails if *stream* is unbuffered, or if *stream*'s buffer needed to be flushed and the function call caused an underlying **spt_writex()** or **lseek()** function to be invoked. In addition, if the **spt_vfprintfx()** function fails, **errno** is set to one of these values:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EILSEQ] An invalid wide character was detected.

[EINTR] The operation was interrupted by a signal that was caught, and no data was transferred.

[EINVAL] There are insufficient arguments.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned. This error might also be returned under implementation-defined conditions.

[ENOMEM] Insufficient storage space was available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

## RELATED INFORMATION

Functions: **fprintf(3)**, **printf(3)**, **sprintf(3)**, **spt_fprintx(2)**, **spt_printfx(2)**, **spt_sprintfx(2)**, **spt_vfprintf(2)**, **spt_vsprintfx(2)**, **vprintf(3)**, **vsprintf(3)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_vprintf** - Initiates thread-aware **vprintf( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**int spt_vprintf(**
**const char ****format*,
**va_list** *printarg***);**

## PARAMETERS

See the **vprintf(3)** reference page either online or in the *Open System Services Library Calls Reference manual*.

## DESCRIPTION

This is a thread-aware version of the **vprintf( )** function. The file descriptor underlying standard output must be nonblocking for this function to be thread-aware.

The following macro maps **spt_vprintf( )** to **vprintf( )** and has been defined in **spthread.h**:

**#define vprintf(format, printarg) spt_vprintf(format, printarg)**

This macro is available only when **SPT_THREAD_AWARE** has been defined before including **spthread.h**, as follows:

**#define SPT_THREAD_AWARE**

## RETURN VALUES

See the **vprintf(3)** reference page. The following also applies:

- The value of **errno** is never set to [EAGAIN] or [EWOULDBLOCK].
- If the file descriptor underlying stdout becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_vprintfx** - Formats a variable number of parameters for output (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <stdarg.h>**]
[**#include <stdio.h>**]
**#include <spthread.h>**

```
int spt_vprintfx (
        const char *format,
        va_list printarg
        );
```

## PARAMETERS

*format* Specifies a character string that contains two types of objects:

- Plain characters, which are copied to the output stream.
- Conversion specifications, each of which causes zero or more items to be fetched from the **stdarg** parameter lists.

*printarg* Specifies the parameters to be printed.

## DESCRIPTION

The **spt_vprintfx( )** function is the thread-aware version of the **vprintf( )** function.

The **spt_vprintfx( )** function formats and writes **stdarg** parameter lists.

This function is the same as the **spt_printfx( )** function, except that it is not called with a variable number of parameters. Instead, it is called with a parameter list pointer as defined by **stdarg**.

## RETURN VALUES

Upon successful completion, this function returns the number of bytes in the output string. Otherwise, a negative value is returned.

If the file descriptor underlying **stdout** becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## ERRORS

The **spt_vprintfx( )** function fails if the standard output stream is unbuffered, or if the buffer needed to be flushed and the function call caused an underlying **spt_writex( )** or **lseek( )** function to be invoked. In addition, if the **spt_vprintfx( )** function fails, **errno** is set to one of these values:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying the output stream and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying the output stream is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

[EILSEQ] An invalid wide character was detected.

[EINTR] The operation was interrupted by a signal that was caught, and no data was transferred.

[EINVAL] There are insufficient arguments.

[EIO] The implementation supports job control; the process is a member of a background process group attempting to write to its controlling terminal; **TOSTOP** is set; the process is neither ignoring nor blocking **SIGTTOU**; and the process group of the process is orphaned. This error might also be returned under implementation-defined conditions.

[ENOMEM] Insufficient storage space was available.

[ENOSPC] No free space was remaining on the device containing the file.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

## RELATED INFORMATION

Functions: **fprintf(3)**, **printf(3)**, **sprintf(3)**, **spt_fprintfx(2)**, **spt_printfx(2)**, **spt_vprintf(2)**, **vfprintf(3)**, **vprintf(3)**, **vsprintf(3)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_waitpid -** Initiates thread-aware **waitpid( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**pid_t spt_waitpid(**
    **pid_t** *pid***,**
    **int** *****stat_loc*,
    **int** *options***);**

## PARAMETERS

See the **waitpid(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **waitpid( )** function. The socket must be nonblocking for this function to be thread-aware.

In **spthread.h**, a mapping of **waitpid( )** to **spt_waitpid( )** has been defined:

```
#define waitpid(pid, stat_loc, options) \
        spt_waitpid(pid, stat_loc, options)
```

For C applications that do not use the nonblocking feature, this mapping is available only when the correct preprocessor has been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE
#include <spthread.h>
```

For C applications that use the nonblocking feature, this mapping is available only when the correct preprocessor has been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE_NONBLOCK
#include <spthread.h>
```

For C++ applications that do not use the nonblocking feature, this mapping is available only when the correct preprocessor has been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE_PRAGMA
#include <spthread.h>
```

For C++ applications that use the nonblocking feature, this mapping is available only when the correct preprocessor has been defined before including **spthread.h**, as follows:

```
#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK
#include <spthread.h>
```

## NOTES

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.

- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

**RETURN VALUES**

See the **waitpid(2)** reference page. Also, if a signal is received via the **pthread_kill(2)** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_wakeup -** Wakes up a thread awaiting tagged I/O

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**extern spt_error_t spt_wakeup(**
    **const short** *filenum***,**
    **const long** *tag***,**
    **const long** *count_transferred***,**
    **const long** *error***);**

## PARAMETERS

*filenum* Specifies the Guardian file number being waited on

*tag* Specifies the tag that is being awaited; the value -1 indicates all tags

*count_transferred*
Specifies byte transfer count of completed I/O

*error* Specifies Guardian error number for IO

## DESCRIPTION

This function wakes up a thread awaiting the tagged I/O on the file with the specified Guardian file number. The awakened thread returns from its call to the **spt_awaitio( )** function with a return value of **SPT_SUCCESS**.

## RETURN VALUES

**SPT_SUCCESS**
One of the following conditions exists:

- *tag* was not -1 and waiting I/O was awakened. Note that only one awaiting I/O was awakened.
- *tag* was -1 and awaiting I/O (if any) was awakened.

**SPT_ERROR** One of the following conditions exists:

The value specified for *filenum* was less than 0 (zero).

- *tag* was not -1 and no awaiting IO was found.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

**NAME**

**spt_write -** Initiates thread-aware **write( )** function

**LIBRARY**

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

**SYNOPSIS**

**#include <spthread.h>**

**ssize_t spt_write(**
**int** *filedes***,**
**void ****buffer***,**
**size_t** *nbytes***);**

**PARAMETERS**

See the **write(2)** reference page.

**DESCRIPTION**

This is a thread-aware version of the **write( )** function. The file descriptor must be nonblocking for this function to be thread-aware.

For C applications, a macro to map **write( )** to **spt_write( )** is available when you use the **#define SPT_THREAD_AWARE** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **write( )** to **spt_write( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

**RETURN VALUES**

See the **write(2)** reference page. The following also applies:

- The value of **errno** is never set to [EWOULDBLOCK] or [EAGAIN].
- If the file descriptor becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 Version 2 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_WRITEREADX -** Writes data to a Guardian file from an array and waits for data to be read back from the file

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_WRITEREADX (**
  **short** *filenum***,**
  **char ****buffer***,**
  **unsigned short** *write_count***,**
  **unsigned short** *read_count***,**
  [**unsigned short ****count_read* ]**,**
  [**long** *tag* ]
  **);**

## PARAMETERS

*filenum* specifies the file number of a Guardian file open instance that identifies the file to be read.

*buffer* specifies an array in the application process in which the information to be written to the file is stored before the call. On return, *buffer* contains the information read from the file.

*write_count* specifies the number of bytes to be written.

*read_count* specifies the number of bytes to be read.

*count_read* is for waited I/O only. This parameter returns a count of the number of bytes returned from the file into *buffer*.

*tag* is for nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **SPT_WRITEREADX( )** function is the thread-aware version of the Guardian WRITEREADX procedure.

The **SPT_WRITEREADX( )** function writes data to a file from an array in the application process, then waits for data to be transferred back from the file. The data from the read portion returns in the same array used for the write portion.

If the file is opened for nowait I/O, you must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This condition also applies to other processes that might be sharing the segment. The application must ensure that the buffer used in the call to the **SPT_WRITEREADX( )** function is not reused before the I/O completes with a call to AWAITIOX.

For programming information about the WRITEREADX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

## Considerations

Buffer use
: **SPT_WRITEREADX( )** is intended for use with 32-bit extended addresses. Therefore, the data buffer for **SPT_WRITEREADX( )** can be either in the caller's stack segment or any extended data segment.

Terminals
: A special hardware feature is incorporated in the asynchronous multiplexer controller that ensures the system is ready to read from the terminal as soon as the write is completed.

Interprocess communication
: The **SPT_WRITEREADX( )** function is used to originate a message to another process that was previously opened, then waits for a reply from that process.

Waited I/O read operation
: If a waited I/O **SPT_WRITEREADX( )** call is executed, the *count_read* parameter indicates the number of bytes actually read.

Nowait I/O read operation
: If a nowait I/O **SPT_WRITEREADX( )** call is executed, *count_read* has no meaning and can be omitted. The count of the number of bytes read is obtained when the I/O operation completes through the *count-transferred* parameter of the Guardian AWAITIOX procedure.

  The **SPT_WRITEREADX( )** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

  Do not change the contents of the data buffer between the initiation and completion of a nowait **SPT_WRITEREADX( )** operation. A retry can copy the data again from the user buffer and cause the wrong data to be written. Avoid sharing a buffer between a **SPT_WRITEREADX( )** and another I/O operation because the contents of the data buffer might change before the write is completed.

Carriage return/line feed sequence after the write
: No carriage return and line feed sequence is sent to the terminal after the write part of the operation.

Location of *buffer* and *count_read*
: The buffer and count transferred can be in the user stack or in an extended data segment. The *buffer* and *count_read* cannot be in the user code space.

  If the *buffer* and *count_read* are in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **SPT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This restriction also applies to other processes that might share the segment. It is the application's responsibility to ensure this.

- If you initiated the I/O with **SPT_WRITEREADX( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- A selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- You can call **SPT_CANCEL( )** or CANCELREQ to cancel nowait I/O initiated with **SPT_WRITEREADX( )**. The I/O is canceled if the file is closed before the I/O completes or if you call the Guardian AWAITIOX procedure with a positive time limit and specific file number and the request times out.

Bounds checking

If the extended address of *buffer* is odd, bounds checking rounds the address to the next lower word boundary and also checks an extra byte. The odd address is used for the transfer.

**RETURN VALUES**

The **SPT_WRITEREADX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

**ERRORS**

None. This function does not set the **errno** variable.

**RELATED INFORMATION**

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_WRITEUPDATEUNLOCKX -** Performs random processing of records in a disk file

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zsptsrl**
H-series OSS processes: **/G/system/zdll*nnn*/zsptdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_WRITEUPDATEUNLOCKX (**
  **short** *filenum***,**
  **char ****buffer***,**
  **unsigned short** *write_count***,**
  [**unsigned short ****count_written*]**,**
  [**long** *tag* ]
  **);**

## PARAMETERS

*filenum* specifies the file number of a Guardian file open instance that identifies the file to be written.

*buffer* specifies an array in the application process in which the information to be written to the file is stored before the call.

*write_count* specifies the number of bytes to be written.

*count_written* returns a count of the number of bytes written to the file from *buffer*.

*tag* is for nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **SPT_WRITEUPDATEUNLOCKX( )** function is the thread-aware version of the Guardian WRITEUPDATEUNLOCKX procedure.

The **SPT_WRITEUPDATEUNLOCKX( )** function performs random processing of records in a Guardian disk file. **SPT_WRITEUPDATEUNLOCKX( )** has two purposes:

- To alter, then unlock, the contents of the record at the current position
- To delete the record at the current position in a key-sequenced or relative file

A call to **SPT_WRITEUPDATEUNLOCKX( )** is equivalent to a call to **SPT_WRITEUPDATEX( )** followed by a call to **SPT_UNLOCKREC( )**. However, the **SPT_WRITEUPDATEUNLOCKX( )** function requires less system processing than do the separate calls to **SPT_WRITEUPDATEX( )** and **SPT_UNLOCKREC( )**.

For programming information about the WRITEUPDATEUNLOCKX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

## Considerations

Buffer use
: **SPT_WRITEUPDATEUNLOCKX( )** is intended for use with 32-bit extended addresses. Therefore, the data buffer for **SPT_WRITEUPDATEUNLOCKX( )** can be either in the caller's stack segment or any extended data segment.

Nowait I/O and **SPT_WRITEUPDATEUNLOCKX( )**
: The **SPT_WRITEUPDATEUNLOCKX( )** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O.

For files audited by the Transaction Management Facility (TMF), You must call the AWAITIOX procedure to complete the **SPT_WRITEUPDATEUNLOCKX( )** operation before **SPT_ENDTRANSACTION( )** or **SPT_ABORTTRANSACTION( )** is called.

Do not change the contents of the data buffer between the initiation and completion of a nowait write operation. A retry can copy the data again from the user buffer and cause the wrong data to be written. Avoid sharing a buffer between a write and another I/O operation because this creates the contents of the write buffer might change before the write is completed.

Random processing and **SPT_WRITEUPDATEUNLOCKX( )**
: For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. Positioning for **SPT_WRITEUPDATEUNLOCKX( )** is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to **SPT_WRITEUPDATEUNLOCKX( )** is rejected with Guardian file-system error 11 (`record does not exist`).

Unstructured files (pointers unchanged)
: For unstructured files, data is written in the position indicated by the current-record pointer. A call to **SPT_WRITEUPDATEUNLOCKX( )** for an unstructured file typically follows a call to the Guardian POSITION procedure or **SPT_READUPDATEX( )**. The current-record and next-record pointers are not changed by a call to **SPT_WRITEUPDATEUNLOCKX( )**.

How **SPT_WRITEUPDATEUNLOCKX( )** works
: The record unlocking performed by **SPT_WRITEUPDATEUNLOCKX( )** functions in the same manner as **SPT_UNLOCKREC( )**.

`Record does not exist`
: Positioning for **SPT_WRITEUPDATEUNLOCKX( )** is always to the record described by the exact value of the current key and current-key specifier. Therefore, if such a record does not exist, the call to **SPT_WRITEUPDATEUNLOCKX( )** is rejected with Guardian file-system error 11.

Invalid write operations to queue files
: DP2 rejects **SPT_WRITEUPDATEUNLOCKX( )** operations with a Guardian file-system error 2.

Location of *buffer* and *count_written*
: The buffer and count transferred can be in the user stack or in an extended data segment. The *buffer* and *count_written* cannot be in the user code space.

If the *buffer* and *count_written* are in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **SPT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This restriction also applies to other processes that might be sharing the segment. It is the application's responsibility to ensure this.
- If you initiated the I/O with **SPT_WRITEUPDATEUNLOCKX( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- A selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- Nowait I/O initiated with **SPT_WRITEUPDATEUNLOCKX( )** can be canceled with a call to **SPT_CANCEL( )** or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or if the Guardian AWAITIOX procedure is called with a positive time limit and specific file number and the request times out.

Bounds checking

If the extended address of *buffer* is odd, bounds checking rounds the address to the next lower word boundary and also checks an extra byte. The odd address is used for the transfer.

All considerations for **SPT_WRITEUPDATEX( )** also apply to this call.

### Use on OSS Objects

This procedure operates only on Guardian objects. If an OSS file is specified, Guardian file-system error 2 occurs.

## RETURN VALUES

The **SPT_WRITEUPDATEUNLOCKX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEX(2)**, **SPT_WRITEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**SPT_WRITEUPDATEX -** Transfers data from an array in the application program to a Guardian file

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <cextdecs.h>**]
**#include <spthread.h>**

**short SPT_WRITEUPDATEX (**
**short** *filenum***,**
**char ****buffer***,**
**unsigned short** *write_count***,**
[**unsigned short ****count_written* ]**,**
[**long** *tag* ]
**);**

## PARAMETERS

*filenum* specifies the file number of a Guardian file open instance that identifies the file to be written.

*buffer* specifies an array in the application process in which the information to be written to the file is stored before the call.

*write_count* specifies the number of bytes to be written.

*count_written* returns a count of the number of bytes written to the file from *buffer*.

*tag* is for nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **SPT_WRITEUPDATEX( )** function is the thread-aware version of the Guardian WRITEUPDATEX procedure.

The **SPT_WRITEUPDATEX( )** function performs random processing of records in a Guardian disk file. **SPT_WRITEUPDATEX( )** has two purposes:

- To alter the contents of the record at the current position
- To delete the record at the current position in a key-sequenced or relative file

Data from the application process's array is written in the position indicated by the setting of the current-record pointer. A call to this procedure typically follows a corresponding call to the **SPT_READX( )** or **SPT_READUPDATEX( )** function. The current-record and next-record pointers are not affected by the **SPT_WRITEUPDATEX( )** procedure.

For magnetic tapes, **SPT_WRITEUPDATEX( )** is used to replace a record in an already written tape. The tape is backspaced one record; the data from the application process's array is written in that area.

For programming information about the WRITEUPDATEX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

**Considerations**

Buffer use
: **SPT_WRITEUPDATEX( )** is intended for use with 32-bit extended addresses. Therefore, the data buffer for **SPT_WRITEUPDATEX( )** can be either in the caller's stack segment or any extended data segment.

I/O counts with unstructured files
: Unstructured files are transparently blocked using one of the four valid block sizes (512, 1024, 2048, or 4096 bytes; 4096 is the default). This transparent block size, known as BUFFERSIZE, is the transfer size used against an unstructured file. While BUFFERSIZE does not change the maximum unstructured transfer (4096 bytes), multiple I/O operations might be performed to satisfy a user's request depending on the BUFFERSIZE chosen. For example, if BUFFERSIZE is 512 bytes, and a request is made to write 4096 bytes, at least eight transfers, each 512 bytes long, will be made. More than eight transfers happen, in this case, if the requested transfer does not start on a BUFFERSIZE boundary.

DP2 performance with unstructured files is best when requested transfers begin on BUFFERSIZE boundaries and are integral multiples of BUFFERSIZE.

Because the maximum blocksize for DP2 structured files is also 4096 bytes, this is also the maximum structured transfer size for DP2.

Deleting locked records
: Deleting a locked record implicitly unlocks that record unless the file is audited, in which case the lock is not removed until the transaction terminates.

Waited **SPT_WRITEUPDATEX( )** calls
: If a waited **SPT_WRITEUPDATEX( )** call is executed, the *count_written* parameter indicates the number of bytes actually written.

Nowait **SPT_WRITEUPDATEX( )** calls
: If a nowait **SPT_WRITEUPDATEX( )** call is executed, *count_written* has no meaning and can be omitted. The count of the number of bytes written is obtained through the *count-transferred* parameter of the Guardian AWAITIOX procedure when the I/O completes.

The **SPT_WRITEUPDATEX( )** procedure must finish with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait I/O. For files audited by the Transaction Management Facility (TMF), the AWAITIOX procedure must be called before the **SPT_ENDTRANSACTION( )** or **SPT_ABORTTRANSACTION( )** function is called.

Do not change the contents of the data buffer between the initiation and completion of a nowait write operation. A retry can copy the data again from the user buffer and cause the wrong data to be written. Avoid sharing a buffer between a write and another I/O operation because the contents of the write buffer might change before the write is completed.

Invalid write operations to queue files
: Attempts to perform **SPT_WRITEUPDATEX( )** operations are rejected with a Guardian file-system error 2.

**Disk File Considerations**

Large data transfers
: To enable large data transfers (more than 4096 bytes), you can use **SPT_SETMODE( )** function 141. See the description of SETMODE functions in the *Guardian Procedure Calls Reference Manual*.

Random processing and **SPT_WRITEUPDATEX()**

For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. Positioning for **SPT_WRITEUPDATEX()** is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to **SPT_WRITEUPDATEX()** is rejected with Guardian file-system error 11 (`record does not exist`).

`File is locked`

If a call to **SPT_WRITEUPDATEX()** is made and the file is locked through a file number other than that supplied in the call, the call is rejected with Guardian file-system error 73 (`file is locked`).

When the just-read record is updated

A call to **SPT_WRITEUPDATEX()** following a call to **SPT_READX()**, without intermediate positioning, updates the record just read.

Unstructured files

Transferring disk file data

If the **SPT_WRITEUPDATEX()** call is to an unstructured disk file, data is transferred to the record location specified by the current-record pointer.

File pointers after a successful call

After a successful **SPT_WRITEUPDATEX()** call to an unstructured file, the current-record and next-record pointers are unchanged.

Number of bytes written

If the unstructured file is created with the odd unstructured attribute (also known as ODDUNSTR) set, the number of bytes written is exactly the number specified in *write_count*. If the odd unstructured attribute is not set when the file is created, the value of *write_count* is rounded up to an even value before the **SPT_WRITEUPDATEX()** call is executed.

You set the odd unstructured attribute with the Guardian FILE_CREATE_, FILE_CREATELIST_, or CREATE procedure, or with the File Utility Program (FUP) SET and CREATE commands.

Structured files

Calling **SPT_WRITEUPDATEX()** after KEYPOSITION

If the call to **SPT_WRITEUPDATEX()** immediately follows a call to the Guardian KEYPOSITION procedure in which a nonunique alternate key is specified as the access path, the **SPT_WRITEUPDATEX()** call fails. A subsequent call to the Guardian FILE_GETINFO_ or FILEINFO procedure shows that Guardian file-system error 46 (`invalid key`) occurred. However, if an intermediate call to **SPT_READX()** or **SPT_READLOCKX()** is performed, the call to **SPT_WRITEUPDATEX()** is permitted because a unique record is identified.

Specifying *write_count* for entry-sequenced files

For entry-sequenced files, the value of *write_count* must match exactly the *write_count* value specified when the record was originally inserted into the file.

Changing the primary-key of a key-sequenced record

An update to a record in a key-sequenced file cannot alter the value of the primary-key field. To change the primary-key field, you must delete the old record (**SPT_WRITEUPDATEX( )** with *write_count* = 0 [zero]) and insert a new record with the key field changed (**SPT_WRITEX( )**).

Current-state indicators after **SPT_WRITEUPDATEX( )**

After a successful **SPT_WRITEUPDATEX( )** call, the current-state indicators remain unchanged.

The buffer and count transferred can be in the user stack or in an extended data segment. The buffer and count transferred cannot be in the user code space.

If the buffer or count transferred is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **SPT_CANCEL( )** function or the Guardian CANCELREQ procedure.
- You must not modify the buffer before the I/O completes with a call to AWAITIOX. This also applies to other processes that might be sharing the segment. It is the application's responsibility to ensure this.
- If you initiated the I/O with **SPT_WRITEUPDATEX( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- The extended segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- Nowait I/O initiated with **SPT_WRITEUPDATEX( )** can be canceled with a call to the **SPT_CANCEL( )** function or the Guardian CANCELREQ procedure. The I/O is canceled if the file is closed before the I/O completes or AWAITIOX is called with a positive time limit and specific file number and the request times out.

Bounds checking

If the extended address of the buffer is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

### Magnetic Tape Considerations

Supported equipment

**SPT_WRITEUPDATEX( )** is permitted only on the 3202 Controller for the 5103 or 5104 Tape Drives. This function is not supported on any other controller/tape drive combination. **SPT_WRITEUPDATEX( )** is specifically not permitted on the following controller/tape drive pairs:

- 3206 Controller and the 5106 Tri-Density Tape Drive
- 3207 Controller and the 5103 & 5104 Tape Drives
- 3208 Controller and the 5130 & 5131 Tape Drives

Specifying the correct number of bytes written

When **SPT_WRITEUPDATEX( )** is used with magnetic tape, the number of bytes to be written must fit exactly; otherwise, information on the tape can be lost. However, no error indication is given.

Limitation of **SPT_WRITEUPDATEX( )** to the same record

Five is the maximum number of times a **SPT_WRITEUPDATEX( )** call can be executed to the same record on tape.

## RETURN VALUES

The **SPT_WRITEUPDATEX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_writev -** Initiate thread-aware **writev( )** function

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <spthread.h>**

**ssize_t spt_writev(**
**int** *filedes***,**
**struct iovec ***iov***,**
**int** *iov_count***);**

## PARAMETERS

See the **writev(2)** reference page.

## DESCRIPTION

This is a thread-aware version of the **writev( )** function. The file descriptor must be nonblocking for this function to be thread-aware.

For C applications, a macro to map **write( )** to **spt_writev( )** is available when you use the **#define SPT_THREAD_AWARE** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **write( )** to **spt_writev( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

## RETURN VALUES

See the **writev(2)** reference page. The following also applies:

- The value of **errno** is never set to [EWOULDBLOCK] or [EAGAIN].
- If the file descriptor becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF].
- If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## STANDARDS CONFORMANCE

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [ECONNRESET], [EFAULT], [EGUARDIANLOCKED], [EINVAL], [ENETDOWN], [ENOTCONN], [ETIMEDOUT], and [EWRONGID] can be returned.

## NAME

**spt_writevx -** Writes to a file from scattered buffers (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <sys/types.h>**]
[**#include <sys/uio.h>**]
**#include <spthread.h>**

**int spt_writevx (**
    **int** *filedes***,**
    **struct iovec ***iov***,**
    **int** *iov_count*
    **);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **spt_acceptx( )**, **creat( )**, **dup( )**, **spt_dup2x( )**, **spt_fcntlx( )**, **open( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*iov* Points to an **iovec** structure that identifies the buffers containing the data to be written.

*iov_count* Specifies the number of **iovec** structure entries (buffers) pointed to by the *iov* parameter.

## DESCRIPTION

The **spt_writevx( )** function is a thread-aware version of the **writev( )** function.

The **spt_writevx( )** function attempts to write data to the file associated with the *filedes* parameter from the set of buffers pointed to by the *iov* parameter.

The **spt_writevx( )** function performs the same action as the **spt_writex( )** function, but gathers the output data from the *iov_count* buffers specified by the **iovec** structure buffers pointed to by the *iov* parameter.

The **iovec** structure is defined in the **sys/uoi.h** header file and contains entries with these members:

```
caddr_t  iov_base;
int      iov_len;
```

The **iov_base** and **iov_len** members of each **iovec** structure entry specify the base address and length of an area in memory from which data should be written. The **spt_writevx( )** function always writes a complete buffer before proceeding to the next.

With regular files and devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. If this incremented file pointer is greater than the length of the file, the length of the file is set to this file offset. Upon return from the **spt_writevx( )** function, the file pointer is incremented by the number of bytes actually written.

With devices incapable of seeking, writing always takes place starting at the current position. For such devices, the value of the file pointer after a call to the **spt_writevx( )** function is always 0 (zero).

Fewer bytes than requested if the device does not have enough space to satisfy the request. In this case, the number of bytes written is returned. For example, suppose a file has space for 20 more bytes of data before reaching a limit. A write request of 512 bytes returns a value of 20.

The limit can be either the end of the physical medium or the value that has been set by the **ulimit( )** function. The next write of a nonzero number of bytes gives a failure return (except as noted later).

Upon successful completion, the **spt_writevx( )** function returns the number of bytes actually written to the file associated with *filedes*.

If the **O_APPEND** status flag of the file is set, the file offset is set to the end of the file before each write operation.

If the **O_SYNC** status flag of the file is set and *filedes* refers to a regular file, a successful **spt_writevx( )** call does not return until the data is delivered to the underlying hardware (as described in the **open(2)** reference page).

The **O_NONBLOCK** flag is effective only on pipes, FIFOs, sockets, and terminal device files (Telserv or OSSTTY processes).

Write requests to a pipe or FIFO file are handled the same way as write requests to a regular file with these exceptions:

- No file offset is associated with a pipe; therfore, each **spt_writevx( )** request appends to the end of the pipe.

- If the size of the **spt_writevx( )** request is less than or equal to the value of the **PIPE_BUF** system variable, the **spt_writevx( )** function is guaranteed to be atomic. The data is not interleaved with data from other processes doing writes on the same pipe.

- If the size of the **spt_writevx( )** request is greater than the value of the **PIPE_BUF** system variable, the file system attempts to resize the pipe buffer from 2 * **PIPE_BUF** to 65,536 bytes. If the resizing is successful, the file system performs atomic writes of up to 32,768 bytes and can transfer up to 52 kilobytes of data from the pipe buffer on subsequent **spt_readx( )** or **spt_readvx( )** calls by the client.

  If the file system cannot resize the buffer, it continues to use the existing buffer. A second attempt at resizing occurs after approximately a minute.

  Writes of greater than **PIPE_BUF** bytes can have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the **O_NONBLOCK** flag is set.

- If the **O_NONBLOCK** flag is not set, an **spt_writevx( )** request to a full pipe causes the process to block until enough space becomes available to handle the entire request.

- If the **O_NONBLOCK** flag is set, **spt_writevx( )** requests are handled differently:

  — The **spt_writevx( )** function does block the process.

  — **spt_writevx( )** requests for **PIPE_BUF** or fewer bytes either succeed completely and return the number of bytes written, or return the value -1 and set **errno** to [EAGAIN].

  — An **spt_writevx( )** request for greater than **PIPE_BUF** bytes either transfers what it can and returns the number of bytes written, or transfers no data and returns the value -1 with **errno** set to [EAGAIN]. Also, if a request is greater than **PIPE_BUF** bytes and all data previously written to the pipe has been read, **writev( )** transfers at least **PIPE_BUF** bytes.

When you attempt to write to a file descriptor (other than a pipe or a FIFO file) for a special character device (a terminal) that supports nonblocking writes and cannot accept data immediately:

- If the **O_NONBLOCK** flag is clear, the **spt_writevx( )** function blocks until the data can be accepted.

- If the **O_NONBLOCK** flag is set, the **spt_writevx( )** function returns the value -1 and **errno** is set to [EAGAIN].

When you attempt to write to a socket with no space available for data:

- If the **O_NONBLOCK** flag is not set, the **spt_writevx( )** function blocks until space becomes available.
- If the **O_NONBLOCK** flag is set, the **spt_writevx( )** function returns the value -1 and sets **errno** to [EAGAIN]. The **O_NONBLOCK** flag has no effect if space is available.

Upon successful completion, the **spt_writevx( )** function marks the **st_ctime** and **st_mtime** fields of the file for update and clears the set-user-ID and set-group-ID attributes if the file is a regular file.

The **spt_fcntlx( )** function provides more information about record locks.

If it is interrupted by a signal before it writes any data, the **spt_writevx( )** function returns the value -1 with **errno** set to [EINTR]. If it is interrupted by a signal after it has successfully written some data, the **spt_writevx( )** function returns the number of bytes that it has written.

### Use on Guardian Objects

Attempting to write to a Guardian file (that is, a file in **/G**) that is locked causes the **spt_writevx( )** function to return -1 and set **errno** to [EGUARDIANLOCKED].

## NOTES

For C applications, a macro to map **writev( )** to **spt_writevx( )** is available when you use the **#define SPT_THREAD_AWARE_NONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **writev( )** to **spt_writevx( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

## RETURN VALUES

Upon successful completion, the **spt_writevx( )** function returns the number of bytes that were actually written. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

If the file descriptor becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_writevx( )** function sets **errno** to the corresponding value:

[EAGAIN] One of these conditions occurred:

- An attempt was made to write to a file descriptor that cannot accept data, and the **O_NONBLOCK** flag is set.
- A write to a pipe (FIFO file) of **PIPE_BUF** bytes or less is requested, **O_NONBLOCK** is set, and not enough free space is available.
- The **O_NONBLOCK** flag is set on this file, and the process would be delayed in the write operation.

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function (such as **spt_writez( )**) is in progress on a regular file and a function that is process-blocking for regular files (such as **read( )**, **spt_read( )**, or **spt_readx( )**) attempts to begin an I/O operation on the same open file.

[EBADF] The *filedes* parameter is not a valid file descriptor open for writing.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] Part of the *iov* parameter points to a location outside of the allocated address space of the process.

[EFBIG] An attempt was made to write a file that exceeds the maximum file size.

[EGUARDIANLOCKED]
An **spt_writevx( )** operation was attempted to a file in the Guardian file system (that is, a file in **/G**) that is locked.

[EINTR] An **spt_writevx( )** operation was interrupted by a signal before any data was written.

[EINVAL] One of these conditions occurred:

- The file position pointer associated with the file specified by the *filedes* parameter was negative.
- The value of the *iov_count* parameter was less than or equal to 0 (zero), or greater than **IOV_MAX**.
- One of the **iov_len** values in the *iov* array was negative or overflowed a data item of type **ssize_t**.
- The sum of the **iov_len** values in the *iov* array overflowed an integer.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
- A physical I/O error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote node, but communication with the remote node has been lost.

[ENOSPC] No free space is left on the fileset containing the file.

[ENOTCONN] An attempt was made to write to a socket that is not connected to a peer socket.

[ENXIO] One of these conditions occurred:

- The device associated with the file descriptor specified by the *filedes* parameter is a block special device or character special file, and the file pointer is out of range.
- No existing device is associated with the file descriptor specified by the *filedes* parameter.

[EPIPE] One of these conditions occurred:

- An attempt was made to write to a pipe or FIFO file that is not open for reading by any process. A **SIGPIPE** signal is sent if the process is running in the OSS environment.
- An attempt was made to write to a pipe that has only one end open.
- An attempt was made to write to a socket that is shut down or closed.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWRONGID] One of these conditions occurred:

- The process attempted an input or output operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and the backup process took over.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for use of the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Functions: **creat(2)**, **fcntl(2)**, **lseek(2)**, **open(2)**, **pipe(2)**, **socket(2)**, **spt_fcntlx(2)**, **spt_write(2)**, **spt_writev(2)**, **spt_writex(2)**, **ulimit(3)**, **writev(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with these exceptions:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.

- When a signal arrives during a call to the **spt_writevx( )** function, instead of returning an EINTR error to the application, the **spt_writevx( )** retries the I/O operation, except in this case: If the **fork( )** function is called by a signal handler that is running on a thread performing an **spt_writevx( )** call, the **spt_writevx( )** call in the child process returns an EINTR error to the application.

## NAME

**spt_writevz -** Writes to a file from scattered buffers (thread-aware version)

## LIBRARY

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <spthread.h>

int spt_writevz(
        int filedes,
        struct iovec *iov,
        int iov_count);
```

## PARAMETERS

*filedes*
: Specifies an open file descriptor obtained from a successful call to the **spt_acceptx( )**, **creat( )**, **creat64( )**, **dup( )**, **spt_dup2x( )**, **spt_fcntlz( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*iov*
: Points to a **iovec** structure that identifies the buffers containing the data to be written.

*iov_count*
: Specifies the number of **iovec** structure entries (buffers) pointed to by the *iov* parameter.

## DESCRIPTION

The **spt_writevz( )** function is a thread-aware version of the **writev( )** function.

The **spt_writevz( )** function attempts to write data to the file associated with the *filedes* parameter from the set of buffers pointed to by the *iov* parameter.

The **spt_writevz( )** function performs the same action as the **spt_writez( )** function, but gathers the output data from the *iov_count* buffers specified by the **iovec** structure buffers pointed to by the *iov* parameter.

The **iovec** structure is defined in the **sys/uoi.h** header file and contains entries with these members:

```
caddr_t  iov_base;
int      iov_len;
```

The **iov_base** and **iov_len** members of each **iovec** structure entry specify the base address and length of an area in memory from which data should be written. The **spt_writevz( )** function always writes a complete buffer before proceeding to the next.

With regular files and devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. If this incremented file pointer is greater than the length of the file, the length of the file is set to this file offset. Upon return from the **spt_writevz( )** function, the file pointer is incremented by the number of bytes actually written.

With devices incapable of seeking, writing always takes place starting at the current position. For such devices, the value of the file pointer after a call to the **spt_writevz( )** function is always 0 (zero).

Fewer bytes than requested can be written if there is not enough room to satisfy the request. In this case, the number of bytes written is returned. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write request of 512 bytes returns a value of 20. The limit reached can be either the end of the physical medium or the value that has been set by the **ulimit( )** function. The next write of a nonzero number of bytes gives a failure return (except as noted later).

Upon successful completion, the **spt_writevz( )** function returns the number of bytes actually written to the file associated with *filedes*.

If the **O_APPEND** status flag of the file is set, the file offset is set to the end of the file prior to each write.

Write requests to a pipe or FIFO file are handled the same as writes to a regular file with these exceptions:

- No file offset is associated with a pipe; therfore, each **spt_writevz( )** request appends to the end of the pipe.
- If the size of the **spt_writevz( )** request is less than or equal to the value of the **PIPE_BUF** system variable, the **spt_writevz( )** function is guaranteed to be atomic. The data is not interleaved with data from other processes doing writes on the same pipe.
- If the size of the **spt_writevz( )** request is greater than the value of the **PIPE_BUF** system variable, the file system attempts to resize the pipe buffer from 2 * **PIPE_BUF** to 65,536 bytes. If the resizing is successful, the file system performs atomic writes of up to 32,768 bytes and can transfer up to 52 kilobytes of data from the pipe buffer on subsequent **spt_readz( )** or **spt_readvz( )** calls by the client.

  If the file system cannot resize the buffer, it continues to use the existing buffer. A second attempt at resizing occurs after approximately a minute elapses.

  Writes of greater than **PIPE_BUF** bytes can have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the **O_NONBLOCK** flag is set.
- If the **O_NONBLOCK** flag is not set, a **spt_writevz( )** request to a full pipe causes the process to block until enough space becomes available to handle the entire request.
- If the **O_NONBLOCK** flag is set, **spt_writevz( )** requests are handled differently in these ways:
  - The **spt_writevz( )** function does block the process.
  - **spt_writevz( )** requests for **PIPE_BUF** or fewer bytes either succeed completely and return the number of bytes written, or return the value -1 and set **errno** to [EAGAIN].
  - A **spt_writevz( )** request for greater than **PIPE_BUF** bytes either transfers what it can and returns the number of bytes written, or transfers no data and returns the value -1 with **errno** set to [EAGAIN]. Also, if a request is greater than **PIPE_BUF** bytes and all data previously written to the pipe has been read, **spt_writevz( )** transfers at least **PIPE_BUF** bytes.

When attempting to write to a file descriptor for a special character device (a terminal) that cannot accept data immediately:

- If the **O_NONBLOCK** flag is clear, the **spt_writevz( )** function blocks until the data can be accepted or an error occurs.
- If the **O_NONBLOCK** flag is set, the **spt_writevz( )** function returns the value -1 and **errno** is set to [EAGAIN].

When attempting to write to a socket with no space available for data:

- If the **O_NONBLOCK** flag is not set, the **spt_writevz( )** function blocks until space becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **spt_writevz( )** function returns the value -1 and sets **errno** to [EWOULDBLOCK].

Upon successful completion, the **spt_writevz( )** function marks the **st_ctime** and **st_mtime** fields of the file for update and clears the set-user-ID and set-group-ID attributes if the file is a regular file.

The **fcntl( )** function provides more information about record locks.

If it is interrupted by a signal before it writes any data, the **spt_writevz( )** function returns the value -1 with **errno** set to [EINTR]. If it is interrupted by a signal after it has successfully written some data, the **spt_writevz( )** function returns the number of bytes that it has written.

### Use on Guardian Objects

Attempting to write to a Guardian file (that is, a file in **/G**) that is locked causes the **spt_writevz( )** function to return -1 and set **errno** to [EGUARDIANLOCKED].

## NOTES

For file descriptors for non-regular files, the **spt_writevz( )** function behaves exactly the same as **spt_writevx( )**. For file descriptors for regular files, this is a thread-aware function: if this function must wait for an I/O operation to complete on an open file, this function blocks the thread that called it (instead of the entire process), while it waits for the I/O operation to complete.

This function serializes file operations on an open file. If a thread calls **spt_writevz( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **writev( )** to **spt_writevz( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **writev( )** to **spt_writevz( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

Upon successful completion, the **spt_writevz( )** function returns the number of bytes that were actually written. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **spt_writevz( )** function sets **errno** to the corresponding value:

[EAGAIN] One of these conditions occurred:

- An attempt was made to write to a file descriptor that cannot accept data, and the **O_NONBLOCK** flag is set.
- A write to a pipe (FIFO file) of **PIPE_BUF** bytes or less is requested, **O_NONBLOCK** is set, and not enough free space is available.
- The **O_NONBLOCK** flag is set on this file, and the process would be delayed in the write operation.

[EBADF] The *filedes* parameter is not a valid file descriptor open for writing.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] Part of the *iov* parameter points to a location outside of the allocated address space of the process.

[EFBIG] The application is attempting to write at or beyond the file offset maximum established when the file was opened.

[EGUARDIANLOCKED]
A **spt_writevz( )** operation was attempted to a file in the Guardian file system (that is, a file in **/G**) that is locked.

[EINTR] A **spt_writevz( )** operation was interrupted by a signal before any data was written.

[EINVAL] One of these conditions occurred:

- The file position pointer associated with the file specified by the *filedes* parameter was negative.
- The value of the *iov_count* parameter was less than or equal to 0 (zero), or greater than **IOV_MAX**.
- One of the **iov_len** values in the *iov* array was negative or overflowed a data item of type **ssize_t**.
- The sum of the **iov_len** values in the *iov* array overflowed an integer.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.

- A physical I/O error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOSPC] No free space is left on the fileset containing the file.

[ENOTCONN] An attempt was made to write to a socket that is not connected to a peer socket.

[ENXIO] One of these conditions occurred:

- The device associated with the file descriptor specified by the *filedes* parameter is a block special device or character special file, and the file pointer is out of range.
- No existing device is associated with the file descriptor specified by the *filedes* parameter.

[EPIPE] One of these conditions occurred:

- An attempt was made to write to a pipe or FIFO file that is not open for reading by any process. A **SIGPIPE** signal is sent if the process is running in the OSS environment.
- An attempt was made to write to a pipe that has only one end open.
- An attempt was made to write to a socket that is shut down or closed.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWOULDBLOCK]
The process attempted an operation on a socket for which **O_NONBLOCK** is set, there is no space available, and no error has occurred.

[EWRONGID] One of these conditions occurred:

- The process attempted an input or output operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor but the new processor lacks a resource or system process needed for use of the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific

Guardian file-system error.

**RELATED INFORMATION**

Functions: **creat(2)**, **creat64(2)**, **fcntl(2)**, **lseek(2)**, **lseek64(2)**, **open(2)**, **open64(2)**, **pipe(2)**, **socket(2)**, **spt_fcntlx(2)**, **spt_write(2)**, **spt_writev(2), spt_writex(2), ulimit(3)**, **writev(2)**.

**STANDARDS CONFORMANCE**

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with these exceptions:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.
- When a signal arrives during a call to the **spt_writevz( )** function, instead of returning an EINTR error to the application, the **spt_writevz( )** retries the I/O operation, except in this case: If the **fork( )** function is called by a signal handler that is running on a thread performing an **spt_writevz( )** call, the **spt_writevz( )** call in the child process returns an EINTR error to the application.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [ECONNRESET], [EFAULT], [EGUARDIANLOCKED], [EINVAL], [ENETDOWN], [ENOTCONN], [ETIMEDOUT], and [EWRONGID] can be returned.

## NAME

**SPT_WRITEX -** Writes data from an array in the application program to an open Guardian file

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

```
[#include <cextdecs.h>]
#include <spthread.h>

short SPT_WRITEX (
        short filenum,
        char *buffer,
        unsigned short write_count,
        [unsigned short *count_written ],
        [long tag ]
        );
```

## PARAMETERS

*filenum* specifies the file number of a Guardian file open instance that identifies the file to be written.

*buffer* specifies an array in the application process in which the information to be written to the file is stored before the call.

*write_count* specifies the number of bytes to be written.

*count_written* returns a count of the number of bytes written to the file from *buffer*.

*tag* is for nowait I/O only. The *tag* value you define uniquely identifies the operation associated with this call.

This parameter is supported only for program compatibility; if you provide it, it is ignored.

## DESCRIPTION

The **SPT_WRITEX( )** function is the thread-aware version of the Guardian WRITEX procedure.

This function writes data from an array in the application program to an open Guardian file.

For programming information about the WRITEX procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

### Considerations

Buffer use
**SPT_WRITEX( )** is intended for use with 32-bit extended addresses. Therefore, the data buffer for **SPT_WRITE( )** can be either in the caller's stack segment or any extended data segment.

Waited I/O and **SPT_WRITEX( )** calls
If a waited **SPT_WRITEX( )** call is executed, the *count_written* parameter indicates the number of bytes actually written.

Nowait I/O and **SPT_WRITEX( )** calls
If a nowait **SPT_WRITE ( )** call is executed, *count_written* has no meaning and can be omitted. The count of the number of bytes written is obtained when the I/O operation completes through the *count-transferred* parameter of the Guardian AWAITIOX procedure.

The **SPT_WRITEX( )** function must complete with a corresponding call to the Guardian AWAITIOX procedure when used with a file that is opened for nowait

I/O.

Do not change the contents of the data buffer between the initiation and completion of a nowait write operation. A retry can copy the data again from the user buffer and cause the wrong data to be written. Avoid sharing a buffer between a write and another I/O operation because the contents of the write buffer might change before the write is completed.

**Disk File Considerations**

Large data transfers for unstructured files using default mode

Default mode allows I/O sizes for unstructured files to be as large as 56KB (57,344), excepting writes to audited files, if the unstructured buffer size (or block size) is 4KB (4096). Default mode refers to the mode of the file if **SPT_SETMODE()** function 141 is not invoked.

For an unstructured file with an unstructured buffer size other than 4KB, DP2 automatically adjusts the unstructured buffer size to 4KB, if possible, when an I/O larger than 4KB is attempted. However, this adjustment is not possible for files that have extents with an odd number of pages; in such cases, an I/O over 4KB is not possible. The switch to a different unstructured buffer size will have a transient performance impact, so HP recommends that the size be initially set to 4KB, which is the default. Transfer sizes over 4KB are not supported in default mode for unstructured access to structured files.

Large data transfers using **SPT_SETMODE(141)**

You can use **SPT_SETMODE()** function 141 to enable large data transfers (more than 4096 bytes) for files opened with unstructured access, regardless of unstructured buffer size. When you use **SPT_SETMODE(141)** to enable large data transfers, you can to specify up to 56K (57344) bytes for the *write_count* parameter. See the description of SETMODE functions in the *Guardian Procedure Calls Reference Manual*.

`File is locked`

If you call **SPT_WRITEX()** is made and the file is locked through a file number other than that supplied in the call, the call is rejected with Guardian file-system error 73 (`file is locked`).

Inserting a new record into a file

The **SPT_WRITEX()** function inserts a new record into a file in the position designated by the file's primary key:

Key-sequenced files

The record is inserted in the position indicated by the value in its primary-key field.

Queue files

The record is inserted into a file at a unique location. The disk process sets the timestamp field in the key, which causes the record to be positioned after the other existing records that have the same high-order user key.

If the file is audited, the record is available for read operations when the transaction associated with the write operation commits. If the transaction aborts, the record is never available to read operations. If the file is not audited, the record is available as soon as the write operation finishes successfully. Unlike other key-sequenced files, a write operation to a queue file will never encounter a Guardian file-system error 10 (`duplicate` record) because all queue file records have unique keys generated for them.

Relative files
: After an open or an explicit positioning by its primary key, the record is inserted in the designated position.

  Subsequent **SPT_WRITEX()** calls without intermediate positioning insert records in successive record positions. If -2 is specified in a preceding positioning, the record is inserted in an available record position in the file.

  If -1 is specified in a preceding positioning, the record is inserted following the last position used in the file. An existing record does not have to be in that position at the time of the **SPT_WRITEX()** call.

Entry-sequenced files
: The record is inserted following the last record currently existing in the file.

Unstructured files
: The record is inserted at the position indicated by the current value of the next-record pointer.

If a record is to be inserted into a key-sequenced or relative file and the record already exists, the **SPT_WRITEX()** call fails, and a subsequent call to the Guardian FILE_GETINFO_ or FILEINFO procedure shows that Guardian file-system error 10 occurred.

Structured files

Inserting records into relative or entry-sequenced files
: If the record is inserted into a relative or entry-sequenced file, the file must be positioned currently through its primary key. Otherwise, the **SPT_WRITEX()** call fails, and a subsequent call to the Guardian FILE_GETINFO_ or FILEINFO procedure shows that Guardian file-system error 46 (invalid key) occurred.

Current-state indicators after an **SPT_WRITEX()** call
: After a successful **SPT_WRITEX()** call, the current-state indicators for positioning mode and comparison length remain unchanged.

  For key-sequenced files, the current position and the current primary-key value remain unchanged.

  For relative and entry-sequenced files, the current position is that of the record just inserted and the current primary-key value is set to the value of the record's primary key.

Duplicate record found on insertion request
: When you attempt to insert a record into a key-sequenced file, if a duplicate record is found, the **SPT_WRITEX()** function returns Guardian file-system error 10 (record already exists) or error 71 (duplicate record). If the operation is part of a TMF transaction, the record is locked for the duration of the transaction.

Unstructured files

DP2 BUFFERSIZE rules

DP2 unstructured files are transparently blocked using one of the four valid DP2 blocksizes (512, 1024, 2048, or 4096 bytes; 4096 is the default). This transparent blocksize, known as BUFFERSIZE, is the transfer size used against an unstructured file. While BUFFERSIZE does not change the maximum unstructured transfer (4096 bytes), multiple I/Os can be performed to satisfy a user request depending on the BUFFERSIZE chosen. For example, if BUFFERSIZE is 512 bytes, and a request is made to write 4096 bytes, at least eight transfers, each 512 bytes long, will be made. More than eight transfers happen, in this case, if the requested transfer does not start on a BUFFERSIZE boundary.

DP2 performance with unstructured files is best when requested transfers begin on BUFFERSIZE boundaries and are integral multiples of BUFFERSIZE.

If the **SPT_WRITEX( )** call is to an unstructured disk file, data is transferred to the record location specified by the next-record pointer. The next-record pointer is updated to point to the record following the record written.

Number of bytes written

If an unstructured file is created with the odd unstructured attribute (also known as ODDUNSTR) set, the number of bytes written is exactly the number specified in *write_count*. If the odd unstructured attribute is not set when the file is created, the value of *write_count* is rounded up to an even value before the **SPT_WRITEX( )** is executed.

You set the odd unstructured attribute with the Guardian FILE_CREATE_, FILE_CREATELIST_, or CREATE procedure, or with the File Utility Program (FUP) SET and CREATE commands.

File pointers after an **SPT_WRITEX( )** call

After a successful **SPT_WRITEX( )** call to an unstructured file, the file pointers have these values:

- Current-record pointer is the next-record pointer.
- Next-record pointer is the next-record pointer plus the count written.
- End-of-file (EOF) pointer is the maximum of the EOF pointer or the next-record pointer.

Use on files opened for nowait I/O

- If the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to the Guardian AWAITIOX procedure or is canceled by a call to the **SPT_CANCEL( )** function or the Guardian CANCELREQ procedure.

- You must not modify the buffer before the I/O completes with a call to the Guardian AWAITIOX procedure. This restriction also applies to other processes that might be sharing the segment. It is the application's responsibility to ensure this.
- If you initiated the I/O with **SPT_WRITE( )**, the I/O must be completed with a call to the Guardian AWAITIOX procedure.
- A selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- You can cancel nowait I/O that was initiated with **SPT_WRITEX( )** with a call to **SPT_CANCEL( )** or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or if the Guardian AWAITIOX procedure is called with a positive time limit and specific file number and the request times out.

### Interprocess Communication Consideration

Indication that the destination process is running

If the **SPT_WRITEX( )** call is to another process, successful completion of the **SPT_WRITEX( )** call (or a Guardian AWAITIOX procedure call if nowait) indicates that the destination process is running.

## RETURN VALUES

The **SPT_WRITEX( )** function returns 0 (zero) upon successful completion. Otherwise, this function returns a nonzero Guardian file-system error number that indicates the outcome of the operation.

For information about Guardian file-system error numbers, see the *Guardian Procedure Errors and Messages Manual*.

## ERRORS

None. This function does not set the **errno** variable.

## RELATED INFORMATION

Functions: **SPT_CANCEL(2)**, **SPT_CONTROL(2)**, **SPT_FILE_CLOSE_(2)**, **SPT_FILE_OPEN_(2)**, **SPT_LOCKFILE(2)**, **SPT_LOCKREC(2)**, **SPT_READLOCKX(2)**, **SPT_READUPDATELOCKX(2)**, **SPT_READUPDATEX(2)**, **SPT_READX(2)**, **SPT_SETMODE(2)**, **SPT_UNLOCKFILE(2)**, **SPT_UNLOCKREC(2)**, **SPT_WRITEREADX(2)**, **SPT_WRITEUPDATEUNLOCKX(2)**, **SPT_WRITEUPDATEX(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to the following industry standards:

- IEEE Std 1003.1c-1995, POSIX System Application Program Interface

The use of the header file **spthread.h** is an HP exception to the POSIX standard.

## NAME

**spt_writex -** Writes to a file (thread-aware version)

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zsptsrl**
H-series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

[**#include <sys/types.h>**]
[**#include <unistd.h>**]
**#include <spthread.h>**

```
ssize_t spt_writex (
        int filedes,
        void *buffer,
        size_t nbytes
        );
```

## PARAMETERS

*filedes* — Specifies an open file descriptor obtained from a successful call to the **spt_acceptx( )**, **creat( )**, **dup( )**, **spt_dup2x( )**, **spt_fcntlx( )**, **open( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*buffer* — Identifies the buffer containing the data to be written.

*nbytes* — Specifies the number of bytes to write.

## DESCRIPTION

The **spt_writex( )** function is the thread-aware version of the **write( )** function.

The **spt_writex( )** function attempts to write *nbytes* of data to the file associated with the *filedes* parameter from the buffer pointed to by the *buffer* parameter.

For all regular and non-regular files, if the value of the *nbytes* parameter is 0 (zero) and the value of *filedes* is a valid file descriptor, the **spt_writex( )** function returns 0 (zero).

The appropriate file time fields are updated unless *nbytes* is 0 (zero).

With regular files and devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. If this incremented file pointer is greater than the length of the file, the length of the file is set to this file offset. Upon return from the **spt_writex( )** function, the file pointer is incremented by the number of bytes actually written.

With devices incapable of seeking, writing always takes place starting at the current position. For such devices, the value of the file pointer after a call to the **spt_writex( )** function is always 0 (zero).

Fewer bytes than requested can be written if the device does not have enough space to satisfy the request. In this case, the number of bytes written is returned. For example, a file has space for 20 bytes more data before reaching a limit. A write request of 512 bytes returns a value of 20. The limit can be either the end of the physical medium or the value that has been set by the **ulimit( )** function. The next write of a nonzero number of bytes gives a failure return (except as noted later).

Upon successful completion, the **spt_writex( )** function returns the number of bytes actually written to the file associated with *filedes*. This number is never greater than the value of *nbytes*.

If the **O_APPEND** flag of the file status is set, the file offset is set to the end of the file before each write operation.

If the **O_SYNC** flag of the file status is set and *filedes* refers to a regular file, a successful **spt_writex( )** call does not return until the data is delivered to the underlying hardware (as described in the **open(2)** reference page).

The **O_NONBLOCK** flag is effective only on pipes, FIFOs, sockets, and terminal device files (Telserv and OSSTTY processes).

Write requests to a pipe or a FIFO file are handled the same way as write requests to a regular file with these exceptions:

- No file offset is associated with a pipe; therfore, each **spt_writex( )** request appends to the end of the pipe.
- If the size of the **spt_writex( )** request is less than or equal to the value of the **PIPE_BUF** system variable, the **spt_writex( )** function is guaranteed to be atomic. The data is not interleaved with data from other processes doing writes on the same pipe.
- If the size of the **spt_writex( )** request is greater than the value of the **PIPE_BUF** system variable, the file system attempts to resize the pipe buffer from 2 * **PIPE_BUF** to 65,536 bytes. If the resizing is successful, the file system performs atomic writes of up to 32,768 bytes and can transfer up to 52 kilobytes of data from the pipe buffer on subsequent **spt_readx( )** or **spt_readvx( )** calls by the client.

  If the file system cannot resize the buffer, it continues to use the existing buffer. A second attempt at resizing occurs after approximately a minute.

  Writes of greater than **PIPE_BUF** bytes can have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the **O_NONBLOCK** flag is set.
- If the **O_NONBLOCK** flag is not set, a **spt_writex( )** request to a full pipe causes the process to block until enough space becomes available to handle the entire request.
- If the **O_NONBLOCK** flag is set, **spt_writex( )** requests are handled differently:
  - The **spt_writex( )** function does block the process.
  - **spt_writex( )** requests for **PIPE_BUF** or fewer bytes either succeed completely and return the value of the *nbytes* parameter, or return the value -1 and set **errno** to [EAGAIN].
  - A **spt_writex( )** request for greater than **PIPE_BUF** bytes either transfers what it can and returns the number of bytes written, or transfers no data and returns the value -1 with **errno** set to [EAGAIN]. Also, if a request is greater than **PIPE_BUF** bytes and all data previously written to the pipe has been read, **spt_writex( )** transfers at least **PIPE_BUF** bytes.

When you attempt to write to a file descriptor (other than a pipe or a FIFO file) for a special character device (a terminal) that supports nonblocking writes and cannot accept data immediately:

- If the **O_NONBLOCK** flag is clear, the **spt_writex( )** function blocks until the data can be accepted.
- If the **O_NONBLOCK** flag is set, the **spt_writex( )** function returns the value -1 and **errno** is set to [EAGAIN].

When you attempt to write to a socket with no space available for data:

- If the **O_NONBLOCK** flag is not set, the **spt_writex( )** function blocks until space becomes available.
- If the **O_NONBLOCK** flag is set, the **spt_writex( )** function returns the value -1 and sets **errno** to [EAGAIN]. The **O_NONBLOCK** flag has no effect if space is available.

Upon successful completion, the **spt_writex( )** function marks the **st_ctime** and **st_mtime** fields of the file for update and clears the set-user-ID and set-group-ID attributes if the file is a regular file.

The **spt_fcntlx( )** function provides more information about record locks.

If it is interrupted by a signal before it writes any data, the **spt_writex( )** function returns the value -1 with **errno** set to [EINTR]. If it is interrupted by a signal after it has successfully written some data, the **spt_writex( )** function returns the number of bytes that it has written.

### Use on Guardian Objects

Attempting to write to a Guardian file (that is, a file in **/G**) that is locked causes the **spt_writex( )** function to return -1 and set **errno** to [EGUARDIANLOCKED].

## NOTES

For C applications, a macro to map **write( )** to **spt_writex( )** is available when you use the **#define SPT_THREAD_AWARE_NONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **write( )** to **spt_writex( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_NONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

## RETURN VALUES

Upon successful completion, the **spt_writex( )** function returns the number of bytes that were actually written. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

If the file descriptor becomes invalid (is closed by another thread), -1 is returned with an **errno** value of [EBADF]. If a signal is received via the **pthread_kill( )** function and is not blocked, ignored, or handled, -1 is returned with an **errno** value of [EINTR].

## ERRORS

If any of these conditions occur, the **spt_writex( )** function sets **errno** to the corresponding value:

[EAGAIN] One of these conditions exists:

- An attempt was made to write to a file descriptor that cannot accept data, and the **O_NONBLOCK** flag is set.
- A write to a pipe (FIFO file) of **PIPE_BUF** bytes or less is requested, **O_NONBLOCK** is set, and fewer than *nbytes* of free space are available.
- The **O_NONBLOCK** flag is set on this file, and the process would be delayed in the write operation.

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function (such as **spt_writez( )**) is in progress on a regular file and a function that is process-blocking for regular files (such as **read( )**, **spt_read( )**, or **spt_readx( )**) attempts to begin an I/O operation on the same open file.

[EBADF] The *filedes* parameter does not specify a valid file descriptor open for writing.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.

- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The *buffer* parameter points to a location outside of the allocated address space of the process.

[EFBIG] An attempt was made to write a file that exceeds the maximum file size.

[EGUARDIANLOCKED]
An **spt_writex( )** operation was attempted on a file in the Guardian file system (that is, a file in **/G**) that is locked.

[EINTR] An **spt_writex( )** operation was interrupted by a signal before any data was written.

[EINVAL] One of these conditions occurred:

- The file position pointer associated with the file specified by the *filedes* parameter was negative.
- The value of the *nbytes* parameter is greater than **SSIZE_MAX**.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
- A physical I/O error occurred. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote node, but communication with the remote node has been lost.

[ENOSPC] No free space is left on the fileset containing the file.

[ENOTCONN] An attempt was made to write to a socket that is not connected to a peer socket.

[ENXIO] One of these conditions occurred:

- The device associated with the file descriptor specified by the *filedes* parameter is a block special device or character special file, and the file pointer is out of range.
- No existing device is associated with the file descriptor specified by the *filedes* parameter.

[EPIPE] One of these conditions occurred:

- An attempt was made to write to a pipe or FIFO file that is not open for reading by any process. A **SIGPIPE** signal is sent if the process is running in the OSS environment.

- An attempt was made to write to a pipe that has only one end open.
- An attempt was made to write to a socket that is shut down or closed.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and the backup process took over.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Functions: **creat(2)**, **fcntl(2)**, **lseek(2)**, **open(2)**, **pipe(2)**, **socket(2)**, **spt_fcntlx(2)**, **spt_write(2)**, **ulimit(3)**, **write(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with these exceptions:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.
- When a signal arrives during a call to the **spt_writex( )** function, instead of returning an EINTR error to the application, the **spt_writex( )** retries the I/O operation, except in this case: If the **fork( )** function is called by a signal handler that is running on a thread performing an **spt_writex( )** call, the **spt_writex( )** call in the child process returns an EINTR error to the application.

The POSIX standard allows certain behaviors of **write( )** to be implementer-defined. For an indication of the HP implementation behaviors, see the **write(2)** reference page either online or in the *Open System Services System Calls Reference Manual*.

## NAME

**spt_writez -** Writes to a file (thread-aware version)

## LIBRARY

H-series and J series OSS processes: **/G/system/zdll***nnn***/zsptdll**

## SYNOPSIS

**#include <sys/types.h>** /* optional except for POSIX.1 */
**#include <unistd.h>**
**#include <spthread.h>**

**ssize_t spt_writez(**
    **int** *filedes***,**
    **void ****buffer***,**
    **size_t** *nbytes***);**

## PARAMETERS

*filedes* Specifies an open file descriptor obtained from a successful call to the **spt_acceptx( )**, **creat( )**, **creat64( )**, **dup( )**, **spt_dup2x( )**, **spt_fcntlx( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*buffer* Identifies the buffer containing the data to be written.

*nbytes* Specifies the number of bytes to write.

## DESCRIPTION

The **spt_writez( )** function is a thread-aware version of the **write( )** function.

The **spt_writez( )** function attempts to write *nbytes* of data to the file associated with the *filedes* parameter from the buffer pointed to by the *buffer* parameter.

For all regular and non-regular files, if the value of the *nbytes* parameter is 0 (zero) and the value of *filedes* is a valid file descriptor, the **spt_writez( )** function returns 0 (zero).

The appropriate file time fields are updated unless *nbytes* is 0 (zero).

With regular files and devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. If this incremented file pointer is greater than the length of the file, the length of the file is set to this file offset. Upon return from the **spt_writez( )** function, the file pointer is incremented by the number of bytes actually written.

With devices incapable of seeking, writing always takes place starting at the current position. For such devices, the value of the file pointer after a call to the **spt_writez( )** function is always 0 (zero).

Fewer bytes than requested can be written if there is not enough room to satisfy the request. In this case, the number of bytes written is returned. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write request of 512 bytes returns a value of 20. The limit reached can be either the end of the physical medium or the value that has been set by the **ulimit( )** function. The next write of a nonzero number of bytes gives a failure return (except as noted later).

Upon successful completion, the **spt_writez( )** function returns the number of bytes actually written to the file associated with *filedes*. This number is never greater than the value of *nbytes*.

If the **O_APPEND** flag of the file status is set, the file offset is set to the end of the file prior to each write.

Write requests to a pipe or a FIFO file are handled the same as writes to a regular file with these exceptions:

- No file offset is associated with a pipe; therfore, each **spt_writez( )** request appends to the end of the pipe.
- If the size of the **spt_writez( )** request is less than or equal to the value of the **PIPE_BUF** system variable, the**spt_writez( )** function is guaranteed to be atomic. The data is not interleaved with data from other processes doing writes on the same pipe.
- If the size of the **spt_writez( )** request is greater than the value of the **PIPE_BUF** system variable, the file system attempts to resize the pipe buffer from 2 * **PIPE_BUF** to 65,536 bytes. If the resizing is successful, the file system performs atomic writes of up to 32,768 bytes and can transfer up to 52 kilobytes of data from the pipe buffer on subsequent **spt_readz( )** or **spt_readvz( )** calls by the client.

  If the file system cannot resize the buffer, it continues to use the existing buffer. A second attempt at resizing occurs after approximately a minute elapses.

  Writes of greater than **PIPE_BUF** bytes can have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the **O_NONBLOCK** flag is set.
- If the **O_NONBLOCK** flag is not set, a **spt_writez( )** request to a full pipe causes the process to block until enough space becomes available to handle the entire request.
- If the **O_NONBLOCK** flag is set, **spt_writez( )** requests are handled differently in these ways:
  - The **spt_writez( )** function does block the process.
  - **spt_writez( )** requests for **PIPE_BUF** or fewer bytes either succeed completely and return the value of the *nbytes* parameter, or return the value -1 and set **errno** to [EAGAIN].
  - A **spt_writez( )** request for greater than **PIPE_BUF** bytes either transfers what it can and returns the number of bytes written, or transfers no data and returns the value -1 with **errno** set to [EAGAIN]. Also, if a request is greater than **PIPE_BUF** bytes and all data previously written to the pipe has been read, **spt_writez( )** transfers at least **PIPE_BUF** bytes.

When attempting to write to a file descriptor for a special character device (a terminal) that cannot accept data immediately:

- If the **O_NONBLOCK** flag is clear, the **spt_writez( )** function blocks until the data can be accepted or an error occurs.
- If the **O_NONBLOCK** flag is set, the **spt_writez( )** function returns the value -1 and **errno** is set to [EAGAIN].

When attempting to write to a socket and with no space available for data:

- If the **O_NONBLOCK** flag is not set, the **spt_writez( )** function blocks until space becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **spt_writez( )** function returns the value -1 and sets **errno** to [EWOULDBLOCK].

Upon successful completion, the **spt_writez( )** function marks the **st_ctime** and **st_mtime** fields of the file for update and clears the set-user-ID and set-group-ID attributes if the file is a regular file.

The **spt_fcntlz( )** function provides more information about record locks.

If it is interrupted by a signal before it writes any data, the **spt_writez( )** function returns the value -1 with **errno** set to [EINTR]. If it is interrupted by a signal after it has successfully written some data, the **spt_writez( )** function returns the number of bytes that it has written.

### Use on Guardian Objects

Attempting to write to a Guardian file (that is, a file in **/G**) that is locked causes the **spt_writez( )** function to return -1 and set **errno** to [EGUARDIANLOCKED].

## NOTES

For file descriptors for non-regular files, the **spt_writez( )** function behaves exactly the same as **spt_writex( )**. For file descriptors for regular files, this is a thread-aware function: if this function must wait for an I/O operation to complete on an open file, this function blocks the thread that called it (instead of the entire process), while it waits for the I/O operation to complete.

This function serializes file operations on an open file. If a thread calls **spt_writez( )** to access a file that already has a file operation in progress by a different thread, this thread is blocked until the prior file operation is complete.

For C applications, a macro to map **write( )** to **spt_writez( )** is available when you use the **#define SPT_THREAD_AWARE_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

For C++ applications, an alias to map **write( )** to **spt_writez( )** is available when you use the **#define SPT_THREAD_AWARE_PRAGMA_XNONBLOCK** preprocessor directive before including **spthread.h** or when you use an equivalent compiler command option to compile the application.

To use this function in a threaded application that uses the Standard POSIX Threads library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks:

- Include the **spthread.h** header file in the application.
- Compile the application using the **_SPT_MODEL_** feature test macro or equivalent compiler command option in addition to any other feature test macros in use.
- Link the application to the **zsptdll** library (**/G/system/zdll***nnn***/zsptdll**).

## RETURN VALUES

Upon successful completion, the **spt_writez( )** function returns the number of bytes that were actually written. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **spt_writez( )** function sets **errno** to the corresponding value:

[EAGAIN] One of these conditions exists:

- An attempt was made to write to a file descriptor that cannot accept data, and the **O_NONBLOCK** flag is set.
- A write to a pipe (FIFO file) of **PIPE_BUF** bytes or less is requested, **O_NONBLOCK** is set, and fewer than *nbytes* of free space are available.
- The **O_NONBLOCK** flag is set on this file, and the process would be delayed in the write operation.

[EBADF] The *filedes* parameter does not specify a valid file descriptor open for writing.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The *buffer* parameter points to a location outside of the allocated address space of the process.

[EFBIG] The application is attempting to write at or beyond the file offset maximum established when the file was opened.

[EGUARDIANLOCKED]
A **spt_writez( )** operation was attempted to a file in the Guardian file system (that is, a file in **/G**) that is locked.

[EINTR] A **spt_writez( )** operation was interrupted by a signal before any data was written.

[EINVAL] One of these conditions occurred:

- The file position pointer associated with the file specified by the *filedes* parameter was negative.
- The value of the *nbytes* parameter is greater than **SSIZE_MAX**.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
- A physical I/O error occurred. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOSPC] No free space is left on the fileset containing the file.

[ENOTCONN] An attempt was made to write to a socket that is not connected to a peer socket.

[ENXIO] One of these conditions occurred:

- The device associated with the file descriptor specified by the *filedes* parameter is a block special device or character special file, and the file pointer is out of range.

- No existing device is associated with the file descriptor specified by the *filedes* parameter.

[EPIPE] One of these conditions occurred:

- An attempt was made to write to a pipe or FIFO file that is not open for reading by any process. A **SIGPIPE** signal is sent if the process is running in the OSS environment.
- An attempt was made to write to a pipe that has only one end open.
- An attempt was made to write to a socket that is shut down or closed.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWOULDBLOCK]
The process attempted an operation on a socket for which **O_NONBLOCK** is set, there is no space available, and no error has occurred.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **creat(2)**, **creat64(2)**, **fcntl(2)**, **lseek(2)**, **lseek64(2)**, **open(2)**, **open64(2)**, **pipe(2)**, **socket(2)**, **spt_fcntlx(2)**, **spt_write(2)**, **ulimit(3)**, **write(2)**.

## STANDARDS CONFORMANCE

This function is an extension to the UNIX 98 specification. Interfaces documented on this reference page conform to IEEE Std 1003.1c-1995, POSIX System Application Program Interface, with these exceptions:

- The use of the header file **spthread.h** is an HP exception to the POSIX standard.
- When a signal arrives during a call to the **spt_writez( )** function, instead of returning an EINTR error to the application, the **spt_writez( )** retries the I/O operation, except in this case: If the **fork( )** function is called by a signal handler that is running on a thread performing an **spt_writez( )** call, the **spt_writez( )** call in the child process returns an EINTR error to the application.

The POSIX standard allows certain behaviors of the **write( )** function to be defined by the vendor. For more information, see the **write(2)** reference page.

## NAME

**stat** - Provides information about a file

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <sys/stat.h>

int stat(
        const char *path,
        struct stat *buffer);
```

## PARAMETERS

*path* Points to the pathname identifying the file.

*buffer* Points to a **stat** structure, into which information is placed about the file. The **stat** structure is described in the **sys/stat.h** header file.

## DESCRIPTION

The **stat( )** function obtains information about the file whose name is pointed to by the *path* parameter. Read, write, or execute permission for the named file is not required, but all directories listed in the pathname leading to the file must be searchable.

The file information is written to the area specified by the *buffer* parameter, which is a pointer to a **stat** structure. For J06.11 and later J-series RVUs and H06.22 and later H-series RVUs, the **stat** structure uses this definition from the **sys/stat.h** header file:

```
struct  stat {
          dev_t     st_dev;
          ino_t     st_ino;
          mode_t    st_mode;
          nlink_t   st_nlink;
          unsigned int        st_acl:1;
          unsigned int        __filler_1:7;
          unsigned int        st_fileprivs:8; /* File privileges */
          uid_t     st_uid;
          gid_t     st_gid;
#if _FILE_OFFSET_BITS != 64 || _TANDEM_ARCH_ == 0
          mode_t    st_basemode; /* Permissions with original group perms */
#endif
          dev_t     st_rdev;
          off_t     st_size;
          time_t    st_atime;
          time_t    st_mtime;
          time_t    st_ctime;
#if _FILE_OFFSET_BITS == 64 && _TANDEM_ARCH_ != 0
          mode_t    st_basemode; /* Permissions with original group perms */
#endif
          int64_t   st_reserved[3];
};
```

For J06.10 and earlier J-series RVUs and H06.21 and earlier H-series RVUs, the **stat** structure uses this definition from the **sys/stat.h** header file:

```
struct  stat {
        dev_t     st_dev;
        ino_t     st_ino;
        mode_t    st_mode;
        nlink_t   st_nlink;
        unsigned int        st_acl:1;
        unsigned int        __filler_1:15;
        uid_t     st_uid;
        gid_t     st_gid;
#if _FILE_OFFSET_BITS != 64 || _TANDEM_ARCH_ == 0
        mode_t    st_basemode; /* Permissions with original group perms */
#endif
        dev_t     st_rdev;
        off_t     st_size;
        time_t    st_atime;
        time_t    st_mtime;
        time_t    st_ctime;
#if _FILE_OFFSET_BITS == 64 && _TANDEM_ARCH_ != 0
        mode_t    st_basemode; /* Permissions with original group perms */
#endif
        int64_t   st_reserved[3];
};
```

For a regular file, the **stat( )** function sets the **st_size** field of the **stat** structure to the length of the file and sets the **st_mode** field to indicate the file type. For a symbolic link, the **stat( )** function returns information about the file at the end of the link; no information about the link is returned. (For information about the link, use the **lstat( )** function.)

The **stat( )** function updates any time-related fields associated with the file before writing into the **stat** structure, unless it is a read-only fileset. Time-related fields are not updated for read-only OSS filesets.

The fields in the **stat** structure have these meanings and content:

**st_dev** OSS device identifier for a fileset.

Values for local OSS objects are listed next. Values for local Guardian objects are described in **Use on Guardian Objects**, and values for remote Guardian or OSS objects are described in **Use on Remote Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | ID of device containing directory entry |
| Directory | ID of device containing directory |
| FIFO | ID of special fileset for pipes |
| **AF_UNIX** socket | ID of device containing the fileset in which the socket file was created |
| **/dev/null** | ID of device containing directory entry |

| | |
|---|---|
| **/dev/tty** | ID of device containing directory entry |

**st_ino** File serial number (inode number). The file serial number and OSS device identifier uniquely identify a regular OSS file within an OSS fileset.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | File serial number (unique) |
| Directory | File serial number (unique) |
| FIFO | File serial number (unique) |
| **AF_UNIX** socket | File serial number of the socket file (unique) |
| **/dev/null** | File serial number (unique) |
| **/dev/tty** | File serial number (unique) |

The **st_ino** value for all node entries in **/E** (including the entry for the logical link from the local node name to the root fileset on the local node) is the value for the root fileset on the corresponding node. If normal conventions are followed, this value is always 0 (zero), so entries in **/E** appear to be nonunique. Values for objects on remote nodes are unique only among the values for objects within the same fileset on that node.

**st_mode** File mode. These bits are ORed into the **st_mode** field:

**S_IFMT** File type. This field can contain one of these values:

**S_IFCHR** Character special file.

**S_IFDIR** Directory.

**S_IFIFO** FIFO.

**S_IFREG** Regular file.

**S_IFSOCK** Socket.

For an **AF_UNIX** socket, the user permissions from the inode for the socket are returned for the permission bits. The access flags are also returned from the inode.

**S_IRWXG** Permissions for the owning group, or if the **st_acl** flag is set, permissions for the the **class** ACL entry.

**S_IRWXO** Other class

**S_IRWXU** Owner class

**S_ISGID** Set group ID on execution

**S_ISUID** Set user ID on execution

**S_ISVTX** Sticky bit; used only for directories (not ORed for files in **/G**, the Guardian file system)

**S_TRUST** Indicates that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers when the memory segment containing the buffers is not shared. This flag applies only to loadfiles for a process and only a user with appropriate privileges (the super ID) can set it.

**S_TRUSTSHARED**
Indicates that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers regardless of whether the memory segment containing the buffers is shared. This flag applies only to loadfiles for a process and only a user with appropriate privileges (the super ID) can set it.

Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

**st_nlink** Number of links.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Number of links to the file |
| Directory | Number of links to the directory |
| FIFO | Number of links to the file |
| **AF_UNIX** socket | Number of links to the socket file |
| **/dev/null** | Number of links to the file |
| **/dev/tty** | Number of links to the file |

**st_acl** If set to 1, indicates that the file has optional access control list (ACL) entries. For compatibility with HP-UX, the member name **st_aclv** is provided as alias for **st_acl**. For more information about ACLs, see the **acl(5)** reference page.

**st_fileprivs** File privileges. For information about file privileges see the **setfilepriv(2)** reference page.

**st_uid** User ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | User ID of the file owner |
| Directory | User ID of the file owner |
| FIFO | User ID of the file owner |
| **AF_UNIX** socket | User ID of the creator of the socket file |
| **/dev/null** | User ID of the super ID |
| **/dev/tty** | User ID of the super ID |

**st_gid** Group ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Group ID of the file group |
| Directory | Group ID of the file group |
| FIFO | Group ID of the file group |
| **AF_UNIX** socket | Group ID of the creator of the socket file |
| **/dev/null** | Group ID of the super ID |
| **/dev/tty** | Group ID of the super ID |

**st_basemode** If the **st_acl** flag is set, contains the permissions for the file owner, owning group, and others. If the **st_acl** flag is not set, **st_basemode** is 0 (zero).

**st_rdev** Remote device ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Undefined |
| Directory | Undefined |
| FIFO | Undefined |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | Undefined |
| **/dev/tty** | ID of the device |

**st_size** File size.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Size of the file in bytes |
| Directory | 4096 |
| FIFO | 0 (zero) |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | 0 (zero) |
| **/dev/tty** | 0 (zero) |

**st_atime** Access time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Time of the last access |
| Directory | Time of the last access |
| FIFO | Time of the last access |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_mtime** Modification time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Time of the last data modification |
| Directory | Time of the last modification |
| FIFO | Time of the last data modification |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_ctime** Status change time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last file status change |
| Directory | Time of the last file status change |
| FIFO | Time of the last file status change |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**Use on Guardian Objects**

The **st_dev** and **st_ino** fields of the **stat** structure do not uniquely identify Guardian files (files in **/G**).

The **st_dev** field is unique for **/G**, for each disk volume and for each Telserv process (or other process of subdevice type 30), because each of these is a separate fileset.

The **S_ISGUARDIANOBJECT** macro can indicate whether an object is a Guardian object when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is a Guardian object and **FALSE** otherwise.

The **st_ino** field is a nonunique encoding of the Guardian filename.

The **st_rdev** field contains a unique minor device number for each **pty***n* entry in **/G/ztnt/**, representing each Telserv process subdevice.

The **st_size** field of an EDIT file (file code 101) is the actual (physical) end of file, not the number of bytes in the file. For directories, **st_size** is set to 4096.

When an OSS function is called for a Guardian EDIT file, the **st_mtime** field is set to the last modification time. The **st_atime** field indicates the last time the file was opened, and the **st_ctime** field is set equal to **st_mtime**. No other time-related fields are updated by OSS function calls.

The **st_ctime** and **st_atime** fields for Guardian regular disk files (except for EDIT files) are updated by OSS function calls, not by Guardian procedure calls.

The time fields for **/G**, **/G/***vol*, and **/G/***vol***/***subvol* always contain the current time.

When the *path* parameter points to the name of a Guardian process that is not a process of subtype 30, the **stat( )** function call fails. The value -1 is returned, and **errno** is set to [ENOENT].

The **stat( )** function always returns access modes of "d---------" when the *path* parameter points to a Guardian subvolume that has a reserved name beginning with ZYQ. The other access modes reported for files in **/G** vary according to the file type.

The next table shows the mapping between Guardian files and their corresponding file types described in the **st_mode** field.

| Example in /G | Guardian File Type | st_mode File Type | Permissions |
|---|---|---|---|
| **/G** | N/A | Directory | r-xr-xr-x |
| *vol* | Disk volume | Directory | rwxrwxrwx |
| *vol*/*subvol* | Subvolume | Directory | rwxrwxrwx |
| *vol*/*subvol*/*fileid* | Disk file | Regular file | See following text |
| *vol*/**#123** | Temporary disk file | Regular file | See following text |
| **ztnt** | Subtype 30 process | Directory | --x--x--x |
| **ztnt/#pty0001** | Subtype 30 process with qualifier | Character special | rw-rw-rw- |
| **vol1/zyq00001** | Subvolume | Directory | --------- |

A Guardian file classified as a directory is always owned by the super ID.

Guardian permissions are mapped as follows:

- Guardian network or any user permission is mapped to OSS other permission.
- Guardian community or group user permission is mapped to OSS group permission.
- Guardian user or owner permission is mapped to OSS owner permission.
- Guardian super ID permission is mapped to OSS super ID permission.
- Guardian read permission is mapped to OSS read permission.
- Guardian write permission is mapped to OSS write permission.
- Guardian execute permission is mapped to OSS execute permission.
- Guardian purge permission is ignored.

Users are not allowed read access to Guardian processes.

OSS file permissions are divided into three groups (owner, group, and other) of three permission bits each (read, write, and execute). The OSS permission bits do not distinguish between remote and local users as Guardian security does; local and remote users are treated alike.

## Use on Remote Objects

The content of the **st_dev** field of the **stat** structure is unique for each node in **/E** because each of these is a separate fileset. Values for directories within **/E** are the same as described for objects on the local HP NonStop node.

The **S_ISEXPANDOBJECT** macro can indicate whether an object in the **/E** directory is on a remote HP NonStop node when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is on a remote HP NonStop node and **FALSE** otherwise.

## Use From the Guardian Environment

The **stat( )** function belongs to a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. You cannot close these file numbers by calling the Guardian FILE_CLOSE_ procedure.

- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

**NOTES**

For J06.08 and earlier J-series RVUs, H06.19 and earlier H-series RVUs, or G-series RVUs, the OSS Network File System (NFS) cannot access OSS objects that have OSS ACLs that contain optional ACL entries.

For J06.09 and later J-series RVUs and H06.20 and later H-series RVUs, access by the OSS Network File System (NFS) to OSS objects that have OSS ACLs that contain optional ACL entries can be allowed, depending upon the NFSPERMMAP attribute value for the fileset that contains the object. For more information about NFS and ACLs, see the **acl(5)** reference page.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

**RETURN VALUES**

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **stat( )** function sets **errno** to the corresponding value:

[EACCES] Search permission is denied for a component of the pathname pointed to by the *path* parameter.

[EFAULT] Either the *buffer* parameter or the *path* parameter points to a location outside of the allocated address space of the process.

[EFSBAD] The program attempted an operation involving a fileset with a corrupted fileset catalog.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[ENAMETOOLONG]
One of these is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of these conditions exists:

- The file specified by the *path* parameter does not exist.
- The *path* parameter points to an empty string.
- The specified pathname cannot be mapped to a valid Guardian filename.
- The specified pathname points to the name of a Guardian process that is not of subtype 30.
- The *path* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOROOT] One of these conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node, and communication with the remote name server has been lost.

[ENOTDIR] A component of the pathname specified by the *path* parameter is not a directory.

[ENOTSUP] The **path** parameter refers to a file on a logical disk volume administered through the Storage Management Foundation (SMF).

[ENXIO] An invalid device or address was specified during an input or output operation on a special file. One of these events occurred:

- A device was specified that does not exist, or a request was made beyond the limits of the device.
- The fileset containing the requestor's current working directory or root directory is not mounted. This error can occur after failure and restart of an OSS name server process until the fileset has been repaired and remounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EOVERFLOW]
The file size (in bytes) or the file inode number (serial number) cannot be represented correctly in the structure pointed to by the *buffer* parameter.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **chown(2)**, **link(2)**, **lstat(2)**, **lstat64(2)**, **mknod(2)**, **open(2)**, **open64(2)**, **pipe(2)**, **setfilepriv(2)**, **utime(2)**.

Miscellaneous Topics: **acl(5)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- For files other than regular disk files or symbolic links, the **st_size** field of the **stat** structure is set to 0 (zero). For directories, **st_size** is set to 4096.
- The **S_IRWXU**, **S_IRWXG**, **S_IRWXO**, **S_IFMT**, **S_ISVTX**, **S_ISGID**, and **S_ISUID** bits are ORed into the **st_mode** field of the **stat** structure.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [EFAULT], [EFSBAD], [ENOROOT], [ENOTSUP], [ENXIO], and [EOSSNOTRUNNING] can be returned by the **stat( )** function.

## NAME

**stat64 -** Provides information about a file

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series and J-series native Guardian processes: implicit libraries
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <sys/stat.h>

int stat64(
        const char *path,
        struct stat64 *buffer);
```

## PARAMETERS

*path* Points to the pathname identifying the file.

*buffer* Points to a **stat64** structure, into which information is placed about the file. The **stat64** structure is described in the **sys/stat.h** header file.

## DESCRIPTION

The **stat64( )** function is similar to the **stat( )** function except that, in addition to supporting smaller files, the **stat64( )** function supports files larger than approximately 2 gigabytes.

An application can explicitly call this function when you compile the application using the **#define _LARGEFILE64_SOURCE 1** feature test macro or an equivalent compiler command option.

An application call to **stat( )** is automatically mapped to this function you compile the application using the **#define _FILE_OFFSET_BITS 64** feature test macro or an equivalent compiler command option.

The **stat64( )** function obtains information about the file whose name is pointed to by the *path* parameter. Read, write, or execute permission for the named file is not required, but all directories listed in the pathname leading to the file must be searchable.

The file information is written to the area specified by the *buffer* parameter, which is a pointer to a **stat64** structure. For J06.11 and later J-series RVUs and H06.22 and later H-series RVUs, the **stat64** structure uses this definition from the **sys/stat.h** header file:

```
struct  stat64 {
        dev_t      st_dev;
        ino64_t    st_ino;
        mode_t     st_mode;
        nlink_t    st_nlink;
        unsigned int       st_acl:1;
        unsigned int       __filler_1:7;
        unsigned int       st_fileprivs:8; /* File privileges */
        uid_t      st_uid;
        gid_t      st_gid;
        dev_t      st_rdev;
        off64_t    st_size;
        time_t     st_atime;
        time_t     st_mtime;
        time_t     st_ctime;
        mode_t     st_basemode; /* Permissions with original group perms */
        int64_t    reserved[3];
};
```

For J06.10 and earlier J-series RVUs and H06.21 and earlier H-series RVUs, the **stat64** structure uses this definition from the **sys/stat.h** header file:

```
struct  stat64 {
        dev_t      st_dev;
        ino64_t    st_ino;
        mode_t     st_mode;
        nlink_t    st_nlink;
        unsigned int       st_acl:1;
        unsigned int       __filler_1:15;
        uid_t      st_uid;
        gid_t      st_gid;
        dev_t      st_rdev;
        off64_t    st_size;
        time_t     st_atime;
        time_t     st_mtime;
        time_t     st_ctime;
        mode_t     st_basemode; /* Permissions with original group perms */
        int64_t    reserved[3];
};
```

For a regular file, the **stat64( )** function sets the **st_size** field of the **stat64** structure to the length of the file and sets the **st_mode** field to indicate the file type. For a symbolic link, the **stat64( )** function returns information about the file at the end of the link; no information about the link is returned. (For information about the link, use the **lstat64( )** function.)

The **stat64( )** function updates any time-related fields associated with the file before writing into the **stat64** structure, unless it is a read-only fileset. Time-related fields are not updated for read-only OSS filesets.

The fields in the **stat64** structure have these meanings and content:

**st_dev** OSS device identifier for a fileset.

Values for local OSS objects are listed next. Values for local Guardian objects are described in **Use on Guardian Objects**, and values for remote Guardian or OSS objects are described in **Use on Remote Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | ID of device containing directory entry |
| Directory | ID of device containing directory |
| FIFO | ID of special fileset for pipes |
| **AF_UNIX** socket | ID of device containing the fileset in which the socket file was created |
| **/dev/null** | ID of device containing directory entry |
| **/dev/tty** | ID of device containing directory entry |

**st_ino** File serial number (inode number). The file serial number and OSS device identifier uniquely identify a regular OSS file within an OSS fileset.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | File serial number (unique) |
| Directory | File serial number (unique) |
| FIFO | File serial number (unique) |
| **AF_UNIX** socket | File serial number of the socket file (unique) |
| **/dev/null** | File serial number (unique) |
| **/dev/tty** | File serial number (unique) |

The **st_ino** value for all node entries in **/E** (including the entry for the logical link from the local node name to the root fileset on the local node) is the value for the root fileset on the corresponding node. If normal conventions are followed, this value is always 0 (zero), so entries in **/E** appear to be nonunique. Values for objects on remote nodes are unique only among the values for objects within the same fileset on that node.

**st_mode** File mode. These bits are ORed into the **st_mode** field:

**S_IFMT** File type. This field can contain one of these values:

**S_IFCHR** Character special file.

**S_IFDIR** Directory.

**S_IFIFO** FIFO.

**S_IFREG** Regular file.

**S_IFSOCK** Socket.

For an **AF_UNIX** socket, the user permissions from the inode for the socket are returned for the permission bits. The access flags are also returned from the inode.

**S_IRWXG** Permissions for the owning group, or if the **st_acl** flag is set, permissions for the the **class** ACL entry.

**S_IRWXO** Other class

**S_IRWXU** Owner class

**S_ISGID** Set group ID on execution

**S_ISUID** Set user ID on execution

**S_ISVTX** Sticky bit; used only for directories (not ORed for files in **/G**, the Guardian file system)

**S_TRUST** Indicates that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers when the memory segment containing the buffers is not shared. This flag applies only to loadfiles for a process and only a user with appropriate privileges (the super ID) can set it.

**S_TRUSTSHARED**
Indicates that the file does not contain code for an uncooperative process or code to examine or modify I/O buffers. This flag suppresses operating system protection of the buffers regardless of whether the memory segment containing the buffers is shared. This flag applies only to loadfiles for a process and only a user with appropriate privileges (the super ID) can set it.

Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

**st_nlink** Number of links.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Number of links to the file |
| Directory | Number of links to the directory |
| FIFO | Number of links to the file |
| **AF_UNIX** socket | Number of links to the socket file |
| **/dev/null** | Number of links to the file |
| **/dev/tty** | Number of links to the file |

**st_acl** If set to 1, indicates that the file has optional access control list (ACL) entries. For compatibility with HP-UX, the member name **st_aclv** is provided as alias for **st_acl**. For more information about ACLs, see the **acl(5)** reference page.

**st_fileprivs** File privileges. For information about file privileges see the **setfilepriv(2)** reference page.

**st_uid** User ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | User ID of the file owner |
| Directory | User ID of the file owner |
| FIFO | User ID of the file owner |
| **AF_UNIX** socket | User ID of the creator of the socket file |
| **/dev/null** | User ID of the super ID |
| **/dev/tty** | User ID of the super ID |

**st_gid** Group ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Group ID of the file group |
| Directory | Group ID of the file group |
| FIFO | Group ID of the file group |
| **AF_UNIX** socket | Group ID of the creator of the socket file |
| **/dev/null** | Group ID of the super ID |
| **/dev/tty** | Group ID of the super ID |

**st_basemode** If the **st_acl** flag is set, contains the permissions for the file owner, owning group, and others. If the **st_acl** flag is not set, **st_basemode** is 0 (zero).

**st_rdev** Remote device ID.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Undefined |
| Directory | Undefined |
| FIFO | Undefined |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | Undefined |
| **/dev/tty** | ID of the device |

**st_size** File size.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Size of the file in bytes |
| Directory | 4096 |
| FIFO | 0 (zero) |
| **AF_UNIX** socket | 0 (zero) |
| **/dev/null** | 0 (zero) |
| **/dev/tty** | 0 (zero) |

**st_atime** Access time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last access |
| Directory | Time of the last access |
| FIFO | Time of the last access |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_mtime** Modification time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| For | Contains |
|---|---|
| Regular file | Time of the last data modification |
| Directory | Time of the last modification |
| FIFO | Time of the last data modification |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**st_ctime** Status change time.

Values for OSS objects are listed next. Values for Guardian objects are described in **Use on Guardian Objects**, later in this reference page.

| **For** | **Contains** |
|---|---|
| Regular file | Time of the last file status change |
| Directory | Time of the last file status change |
| FIFO | Time of the last file status change |
| **AF_UNIX** socket | Value retrieved from the inode |
| **/dev/null** | Current time |
| **/dev/tty** | Composite value of the times of all openers of the file |

For the **/E** entry of the local node, the value is the time of the most recent mounting of the root fileset.

**Use on Guardian Objects**

The **st_dev** and **st_ino** fields of the **stat64** structure do not uniquely identify Guardian files (files in **/G**).

The **st_dev** field is unique for **/G**, for each disk volume and for each Telserv process (or other process of subdevice type 30), because each of these is a separate fileset.

The **S_ISGUARDIANOBJECT** macro can indicate whether an object is a Guardian object when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is a Guardian object and **FALSE** otherwise.

The **st_ino** field is a nonunique encoding of the Guardian filename.

The **st_rdev** field contains a unique minor device number for each **pty***n* entry in **/G/ztnt/**, representing each Telserv process subdevice.

The **st_size** field of an EDIT file (file code 101) is the actual (physical) end of file, not the number of bytes in the file. For directories, **st_size** is set to 4096.

When an OSS function is called for a Guardian EDIT file, the **st_mtime** field is set to the last modification time. The **st_atime** field indicates the last time the file was opened, and the **st_ctime** field is set equal to **st_mtime**. No other time-related fields are updated by OSS function calls.

The **st_ctime** and **st_atime** fields for Guardian regular disk files (except for EDIT files) are updated by OSS function calls, not by Guardian procedure calls.

The time fields for **/G**, **/G/***vol*, and **/G/***vol***/***subvol* always contain the current time.

When the *path* parameter points to the name of a Guardian process that is not a process of subtype 30, the **stat64( )** function call fails. The value -1 is returned, and **errno** is set to [ENOENT].

The **stat64( )** function always returns access modes of "d---------" when the *path* parameter points to a Guardian subvolume that has a reserved name beginning with ZYQ. The other access modes reported for files in **/G** vary according to the file type.

The next table shows the mapping between Guardian files and their corresponding file types described in the **st_mode** field.

| **Example in /G** | **Guardian File Type** | **st_mode File Type** | **Permissions** |
|---|---|---|---|
| **/G** | N/A | Directory | r-xr-xr-x |
| *vol* | Disk volume | Directory | rwxrwxrwx |
| *vol*/*subvol* | Subvolume | Directory | rwxrwxrwx |
| *vol*/*subvol*/*fileid* | Disk file | Regular file | See following text |
| *vol*/**#123** | Temporary disk file | Regular file | See following text |
| **ztnt** | Subtype 30 process | Directory | --x--x--x |
| **ztnt/#pty0001** | Subtype 30 process with qualifier | Character special | rw-rw-rw- |
| **vol1/zyq00001** | Subvolume | Directory | --------- |

A Guardian file classified as a directory is always owned by the super ID.

Guardian permissions are mapped as follows:

- Guardian network or any user permission is mapped to OSS other permission.
- Guardian community or group user permission is mapped to OSS group permission.
- Guardian user or owner permission is mapped to OSS owner permission.
- Guardian super ID permission is mapped to OSS super ID permission.
- Guardian read permission is mapped to OSS read permission.
- Guardian write permission is mapped to OSS write permission.
- Guardian execute permission is mapped to OSS execute permission.
- Guardian purge permission is ignored.

Users are not allowed read access to Guardian processes.

OSS file permissions are divided into three groups (owner, group, and other) of three permission bits each (read, write, and execute). The OSS permission bits do not distinguish between remote and local users as Guardian security does; local and remote users are treated alike.

### Use on Remote Objects

The content of the **st_dev** field of the **stat64** structure is unique for each node in **/E** because each of these is a separate fileset. Values for directories within **/E** are the same as described for objects on the local HP NonStop node.

The **S_ISEXPANDOBJECT** macro can indicate whether an object in the **/E** directory is on a remote HP NonStop node when the **st_dev** field is passed to the macro. The value of the macro is **TRUE** if the object is on a remote HP NonStop node and **FALSE** otherwise.

### Use From the Guardian Environment

The **stat64( )** function belongs to a set of functions that have these effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. You cannot close these file numbers by calling the Guardian FILE_CLOSE_ procedure.

- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

**NOTES**

For J06.08 and earlier J-series RVUs, H06.19 and earlier H-series RVUs, or G-series RVUs, the OSS Network File System (NFS) cannot access OSS objects that have OSS ACLs that contain optional ACL entries.

For J06.09 and later J-series RVUs and H06.20 and later H-series RVUs, access by the OSS Network File System (NFS) to OSS objects that have OSS ACLs that contain optional ACL entries can be allowed, depending upon the NFSPERMMAP attribute value for the fileset that contains the object. For more information about NFS and ACLs, see the **acl(5)** reference page.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

**RETURN VALUES**

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **stat64( )** function sets **errno** to the corresponding value:

[EACCES] Search permission is denied for a component of the pathname pointed to by the *path* parameter.

[EFAULT] Either the *buffer* parameter or the *path* parameter points to a location outside of the allocated address space of the process.

[EFSBAD] The program attempted an operation involving a fileset with a corrupted fileset catalog.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[ENAMETOOLONG]
One of these is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of these conditions exists:

- The file specified by the *path* parameter does not exist.

- The *path* parameter points to an empty string.
- The specified pathname cannot be mapped to a valid Guardian filename.
- The specified pathname points to the name of a Guardian process that is not of subtype 30.
- The *path* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOROOT] One of these conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node, and communication with the remote name server has been lost.

[ENOTDIR] A component of the pathname specified by the *path* parameter is not a directory.

[ENOTSUP] The **path** parameter refers to a file on a logical disk volume administered through the Storage Management Foundation (SMF).

[ENXIO] An invalid device or address was specified during an input or output operation on a special file. One of these events occurred:

- A device was specified that does not exist, or a request was made beyond the limits of the device.
- The fileset containing the requestor's current working directory or root directory is not mounted. This error can occur after failure and restart of an OSS name server process until the fileset has been repaired and remounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. For more information about a specific Guardian file-system error, see the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Commands: **getacl(1)**, **setacl(1)**.

Functions: **acl(2)**, **chmod(2)**, **chown(2)**, **link(2)**, **lstat(2)**, **lstat64(2)**, **mknod(2)**, **open(2)**, **open64(2)**, **pipe(2)**, **utime(2)**.

Miscellaneous Topics: **acl(5)**.

## STANDARDS CONFORMANCE

This function is an HP extension to the XPG4 Version 2 specification.

## NAME

**statvfs -** Gets fileset information using a pathname

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/statvfs.h>**

**int statvfs(**
**const char ***_path_**,**
**struct statvfs ***_buffer_**);**

## PARAMETERS

_path_ Is a pathname that specifies any file within a mounted fileset.

_buffer_ Points to a **statvfs** structure that is to hold the returned information for the **statvfs( )** call.

## DESCRIPTION

The **statvfs( )** function returns descriptive information about a mounted fileset. The information is returned in a **statvfs** structure, which has the following definition from the **sys/statvfs.h** header file:

```
typedef struct statvfs {
        u_long          f_bsize;
        u_long          f_frsize;
        fsblkcnt_t      f_blocks;
        fsblkcnt_t      f_bfree;
        fsblkcnt_t      f_bavail;
        fsfilcnt_t f_files;
        fsfilcnt_t f_ffree;
        fsfilcnt_t f_favail;
        u_long          f_fsid;
        char            f_basetype[FSTYPSZ];
        u_long          f_flag;
        u_long          f_namemax;
        char            f_fstr[32];
        fsblkcnt_t      f_bminavail;
        fsblkcnt_t      f_bmaxavail;
        u_long          f_filler[5];
} statvfs_t;
```

The fields in this structure have the following meanings and content:

**f_bsize** Fileset block size.

| **For** | **Contains** |
|---|---|
| Regular file | 4096 |
| Directory | 4096 |
| FIFO | 4096 |
| **AF_UNIX** socket | 4096 |
| **/dev/null** | 4096 |
| Object in **/G** | 4096 |
| **/G** | 4096 |
| **/G/ztnt/#pty***nn* | 4096 |
| **/E** | 4096 |

**f_frsize** Fundamental file system block size.

| **For** | **Contains** |
|---|---|
| Regular file | 4096 |
| Directory | 4096 |
| FIFO | 4096 |
| **AF_UNIX** socket | 4096 |
| **/dev/null** | 4096 |
| Object in **/G** | 4096 |
| **/G** | 4096 |
| **/G/ztnt/#pty***nn* | 4096 |
| **/E** | 4096 |

**f_blocks** Total number of blocks in fileset, in units of **f_frsize**.

| **For** | **Contains** |
|---|---|
| Regular file | Number of blocks on all volumes ever used in the fileset. |
| Directory | Number of blocks on all volumes ever used in the fileset. |
| FIFO | Number of blocks on all volumes ever used in the fileset. |
| **AF_UNIX** socket | Number of blocks on all volumes ever used in the fileset. |
| **/dev/null** | Number of blocks on all volumes ever used in the fileset. |
| Object in **/G** | Number of blocks on the volume containing the object. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

**f_bfree** Total number of free blocks in fileset.

| **For** | **Contains** |
|---|---|
| Regular file | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Directory | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| FIFO | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| **AF_UNIX** socket | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| **/dev/null** | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Object in **/G** | Number of free blocks in the volume containing the object. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

**f_bavail** Number of free blocks available to a process without appropriate privileges.

| **For** | **Contains** |
|---|---|
| Regular file | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Directory | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| FIFO | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| **AF_UNIX** socket | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| **/dev/null** | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Object in **/G** | Number of free blocks in the volume containing the object. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |

| | |
|---|---|
| **/E** | 0 |

**f_files** Total number of file serial numbers (inode numbers) in the fileset.

| For | Contains |
|---|---|
| Regular file | Number of inode numbers in the fileset. |
| Directory | Number of inode numbers in the fileset. |
| FIFO | Number of inode numbers in the fileset. |
| **AF_UNIX** socket | Number of inode numbers in the fileset. |
| **/dev/null** | Number of inode numbers in the fileset. |
| Object in **/G** | The value of **ULONG_MAX**. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

**f_ffree** Total number of free file serial numbers (inode numbers) in the fileset.

| For | Contains |
|---|---|
| Regular file | Number of free inode numbers in the fileset. |
| Directory | Number of free inode numbers in the fileset. |
| FIFO | Number of free inode numbers in the fileset. |
| **AF_UNIX** socket | Number of free inode numbers in the fileset. |
| **/dev/null** | Number of free inode numbers in the fileset. |
| Object in **/G** | The value of **ULONG_MAX**. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

**f_favail** Number of file serial numbers (inode numbers) available to a process without appropriate privileges.

| For | Contains |
|---|---|
| Regular file | Number of free inode numbers in the fileset. |
| Directory | Number of free inode numbers in the fileset. |
| FIFO | Number of free inode numbers in the fileset. |
| **AF_UNIX** socket | Number of free inode numbers in the fileset. |

| | |
|---|---|
| **/dev/null** | Number of free inode numbers in the fileset. |
| Object in **/G** | The value of **ULONG_MAX**. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

**f_fsid** Fileset identifier.

| **For** | **Contains** |
|---|---|
| Regular file | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| Directory | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| FIFO | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **AF_UNIX** socket | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/dev/null** | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| Object in **/G** | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/G** | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/G/ztnt/#pty***nn* | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/E** | Lower 32 bits of the **st_dev** field in the **stat** structure. |

**f_basetype** Type of file system.

| **For** | **Contains** |
|---|---|
| Regular file | **OSS** |
| Directory | **OSS** |
| FIFO | **OSS** |
| **AF_UNIX** socket | **OSS** |
| **/dev/null** | **OSS** |
| Object in **/G** | **GUARDIAN** |
| **/G** | **GUARDIAN** |
| **/G/ztnt/#pty***nn* | **GUARDIAN** |
| **/E** | **EXPAND** |

**f_flag** Bit mask indicating type of fileset access allowed.

| **For** | **Contains** |
|---|---|
| Regular file | 4 if fileset is read/write, 5 if fileset is read-only. |
| Directory | 4 if fileset is read/write, 5 if fileset is read-only. |
| FIFO | 4 if fileset is read/write, 5 if fileset is read-only. |
| **AF_UNIX** socket | 4 if fileset is read/write, 5 if fileset is read-only. |
| **/dev/null** | 4 if fileset is read/write, 5 if fileset is read-only. |
| Object in **/G** | 2 |
| **/G** | 3 |
| **/G/ztnt/#pty***nn* | 2 |
| **/E** | 3 |

You can test the content of the **f_flag** field with the following symbolic values:

**ST_NOSUID** This bit flag is set if the fileset does not allow the **setuid** bit to be set for its member files.

**ST_NOTRUNC**
This bit flag is set if the fileset does not truncate filenames.

**ST_RDONLY** This bit flag is set if the fileset is mounted for read-only access.

**f_namemax** Maximum number of character bytes in a filename within the fileset.

| **For** | **Contains** |
|---|---|
| Regular file | 248 |
| Directory | 248 |
| FIFO | 248 |
| **AF_UNIX** socket | 248 |
| **/dev/null** | 248 |
| Object in **/G** | 8 |
| **/G** | 7 |
| **/G/ztnt/#pty***nn* | 7 |
| **/E** | 7 |

**f_fstr** Fileset pathname prefix string.

| **For** | **Contains** |
|---|---|
| Regular file | **/E/***nodename***/G/***volume***/ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| Directory | **/E/***nodename***/G/***volume***/ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| FIFO | **/E/***nodename***/G/***volume***/ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| **AF_UNIX** socket | **/E/***nodename***/G/***volume***/ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| **/dev/null** | **/E/***nodename***/G/***volume***/ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| Object in **/G** | **/E/***nodename***/G/***volume*, identifying the disk volume containing the specified file. |
| **/G** | **/E/***nodename***/G** |
| **/G/ztnt/#pty***nn* | **/E/***nodename***/G** |
| **/E** | **/E** |

**f_bminavail** Number of blocks free on the disk volume with the least space remaining.

| **For** | **Contains** |
|---|---|
| Regular file | Number of blocks. |
| Directory | Number of blocks. |
| FIFO | Number of blocks. |
| **AF_UNIX** socket | Number of blocks. |
| **/dev/null** | Number of blocks. |
| Object in **/G** | Number of blocks. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

**f_bmaxavail** Number of blocks free on the disk volume with the most space remaining.

| **For** | **Contains** |
|---|---|
| Regular file | Number of blocks. |
| Directory | Number of blocks. |
| FIFO | Number of blocks. |
| **AF_UNIX** socket | Number of blocks. |
| **/dev/null** | Number of blocks. |
| Object in **/G** | Number of blocks. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

### Use From the Guardian Environment

The **statvfs( )** function is one of a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. You cannot close these file numbers by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

This function provides compatibility with the System V Interface Definition, Revision 3.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **statvfs( )** function returns the value 0 (zero). Otherwise, it returns the value -1 and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **statvfs( )** function sets **errno** to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix of the *path* parameter.

[EFAULT] The *buffer* or *path* parameter points to a location outside of the allocated address space of the process.

[EINTR] The function was interrupted by a signal before any data arrived.

[EIO] One of the following conditions occurred:

- The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.

- A physical I/O error has occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed. Data might have been lost during a transfer.

[ELOOP] Too many symbolic links were encountered in translating the *path* parameter.

[ENAMETOOLONG]
One of these names is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

You can call the **pathconf( )** function to obtain the applicable limits.

[ENOENT] The file referred to by the *path* parameter does not exist.

[ENOTDIR] A component of the path prefix of the *path* parameter is not a directory.

[EOVERFLOW]
One of the values returned cannot be represented correctly in the structure pointed to by the *buffer* **parameter.**

**RELATED INFORMATION**

Functions: **fstat(2)**, **fstatvfs(2)**, **lstat(2)**, **stat(2)**, **statvfs64(2)**.

## NAME

**statvfs64 -** Gets fileset information using a pathname

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/statvfs.h>**

**int statvfs64(**
  **const char ****path***,**
  **struct statvfs64 ****buffer***);**

## PARAMETERS

*path* Is a pathname that specifies any file within a mounted fileset.

*buffer* Points to a **statvfs64** structure that is to hold the returned information for the **statvfs64( )** call.

## DESCRIPTION

The **statvfs64( )** function is similar to the **statvfs( )** function except that, in addition to supporting smaller files, the **statvfs64( )** function supports files larger than approximately 2 gigabytes.

An application can explicitly call this function when you compile the application using the **#define _LARGEFILE64_SOURCE 1** feature test macro or an equivalent compiler command option.

An application call to **creat( )** is automatically mapped to this function when you compile the application using the **#define _FILE_OFFSET_BITS 64** feature test macro or an equivalent compiler command option.

The **statvfs64( )** function returns descriptive information about a mounted fileset. The information is returned in a **statvfs64** structure, which has the following definition from the **sys/statvfs.h** header file:

```
typedef struct statvfs64 {
        unsigned long           f_bsize;
        unsigned long           f_frsize;
        unsigned long long      f_blocks;
        unsigned long long      f_bfree;
        unsigned long long      f_bavail;
        unsigned long long      f_files;
        unsigned long long      f_ffree;
        unsigned long long      f_favail;
        unsigned long           f_fsid;
        char                    f_basetype[FSTYPSZ];
        unsigned long           f_flag;
        unsigned long           f_namemax;
        char                    f_fstr[32];
        unsigned long           f_bminavail;
        unsigned long           f_bmaxavail;
        unsigned long           f_filler[5];
} statvfs64_t;
```

The fields in this structure have the following meanings and content:

**f_bsize** Fileset block size.

| **For** | **Contains** |
|---|---|
| Regular file | 4096 |
| Directory | 4096 |
| FIFO | 4096 |
| **AF_UNIX** socket | 4096 |
| **/dev/null** | 4096 |
| Object in **/G** | 4096 |
| **/G** | 4096 |
| **/G/ztnt/#pty***nn* | 4096 |
| **/E** | 4096 |

**f_frsize** Fundamental file system block size.

| **For** | **Contains** |
|---|---|
| Regular file | 4096 |
| Directory | 4096 |
| FIFO | 4096 |
| **AF_UNIX** socket | 4096 |
| **/dev/null** | 4096 |
| Object in **/G** | 4096 |
| **/G** | 4096 |
| **/G/ztnt/#pty***nn* | 4096 |
| **/E** | 4096 |

**f_blocks** Total number of blocks in fileset, in units of **f_frsize**.

| **For** | **Contains** |
|---|---|
| Regular file | Number of blocks on all volumes ever used in the fileset. |
| Directory | Number of blocks on all volumes ever used in the fileset. |
| FIFO | Number of blocks on all volumes ever used in the fileset. |
| **AF_UNIX** socket | Number of blocks on all volumes ever used in the fileset. |
| **/dev/null** | Number of blocks on all volumes ever used in the fileset. |
| Object in **/G** | Number of blocks on the volume containing the object. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |

| | |
|---|---|
| **/E** | 0 |

**f_bfree** Total number of free blocks in fileset.

| For | Contains |
|---|---|
| Regular file | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Directory | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| FIFO | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| **AF_UNIX** socket | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| **/dev/null** | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Object in **/G** | Number of free blocks in the volume containing the object. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

**f_bavail** Number of free blocks available to a process without appropriate privileges.

| For | Contains |
|---|---|
| Regular file | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Directory | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| FIFO | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| **AF_UNIX** socket | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| **/dev/null** | Number of free blocks on all volumes currently in the storage-pool file for the fileset. |
| Object in **/G** | Number of free blocks in the volume containing the object. |
| **/G** | 0 |

| | |
|---|---|
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

**f_files** Total number of file serial numbers (inode numbers) in the fileset.

| **For** | **Contains** |
|---|---|
| Regular file | Number of inode numbers in the fileset. |
| Directory | Number of inode numbers in the fileset. |
| FIFO | Number of inode numbers in the fileset. |
| **AF_UNIX** socket | Number of inode numbers in the fileset. |
| **/dev/null** | Number of inode numbers in the fileset. |
| Object in **/G** | The value of **ULONG_MAX**. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

**f_ffree** Total number of free file serial numbers (inode numbers) in the fileset.

| **For** | **Contains** |
|---|---|
| Regular file | Number of free inode numbers in the fileset. |
| Directory | Number of free inode numbers in the fileset. |
| FIFO | Number of free inode numbers in the fileset. |
| **AF_UNIX** socket | Number of free inode numbers in the fileset. |
| **/dev/null** | Number of free inode numbers in the fileset. |
| Object in **/G** | The value of **ULONG_MAX**. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

**f_favail** Number of file serial numbers (inode numbers) available to a process without appropriate privileges.

| **For** | **Contains** |
|---|---|
| Regular file | Number of free inode numbers in the fileset. |
| Directory | Number of free inode numbers in the fileset. |
| FIFO | Number of free inode numbers in the fileset. |
| **AF_UNIX** socket | Number of free inode numbers in the fileset. |

| | |
|---|---|
| **/dev/null** | Number of free inode numbers in the fileset. |
| Object in **/G** | The value of **ULONG_MAX**. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

**f_fsid** Fileset identifier.

| **For** | **Contains** |
|---|---|
| Regular file | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| Directory | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| FIFO | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **AF_UNIX** socket | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/dev/null** | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| Object in **/G** | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/G** | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/G/ztnt/#pty***nn* | Lower 32 bits of the **st_dev** field in the **stat** structure. |
| **/E** | Lower 32 bits of the **st_dev** field in the **stat** structure. |

**f_basetype** Type of file system.

| **For** | **Contains** |
|---|---|
| Regular file | **OSS** |
| Directory | **OSS** |
| FIFO | **OSS** |
| **AF_UNIX** socket | **OSS** |
| **/dev/null** | **OSS** |
| Object in **/G** | **GUARDIAN** |
| **/G** | **GUARDIAN** |
| **/G/ztnt/#pty***nn* | **GUARDIAN** |
| **/E** | **EXPAND** |

**f_flag** Bit mask indicating type of fileset access allowed.

| For | Contains |
|---|---|
| Regular file | 4 if fileset is read/write, 5 if fileset is read-only. |
| Directory | 4 if fileset is read/write, 5 if fileset is read-only. |
| FIFO | 4 if fileset is read/write, 5 if fileset is read-only. |
| **AF_UNIX** socket | 4 if fileset is read/write, 5 if fileset is read-only. |
| **/dev/null** | 4 if fileset is read/write, 5 if fileset is read-only. |
| Object in **/G** | 2 |
| **/G** | 3 |
| **/G/ztnt/#pty***nn* | 2 |
| **/E** | 3 |

You can test the content of the **f_flag** field with the following symbolic values:

**ST_NOSUID** This bit flag is set if the fileset does not allow the **setuid** bit to be set for its member files.

**ST_NOTRUNC** This bit flag is set if the fileset does not truncate filenames.

**ST_RDONLY** This bit flag is set if the fileset is mounted for read-only access.

**f_namemax** Maximum number of character bytes in a filename within the fileset.

| For | Contains |
|---|---|
| Regular file | 248 |
| Directory | 248 |
| FIFO | 248 |
| **AF_UNIX** socket | 248 |
| **/dev/null** | 248 |
| Object in **/G** | 8 |
| **/G** | 7 |
| **/G/ztnt/#pty***nn* | 7 |
| **/E** | 7 |

**f_fstr** Fileset pathname prefix string.

| For | Contains |
|---|---|
| Regular file | **/E/***nodename***/G/***volume***/ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| Directory | **/E/***nodename***/G/***volume***/ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| FIFO | **/E/***nodename***/G/***volume***/ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| **AF_UNIX** socket | **/E/***nodename***/G/***volume***/ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| **/dev/null** | **/E/***nodename***/G/***volume***/ZX***nnnnnn n*, identifying the catalog file and version for the specified file. |
| Object in **/G** | **/E/***nodename***/G/***volume*, identifying the disk volume containing the specified file. |
| **/G** | **/E/***nodename***/G** |
| **/G/ztnt/#pty***nn* | **/E/***nodename***/G** |
| **/E** | **/E** |

**f_bminavail** Number of blocks free on the disk volume with the least space remaining.

| For | Contains |
|---|---|
| Regular file | Number of blocks. |
| Directory | Number of blocks. |
| FIFO | Number of blocks. |
| **AF_UNIX** socket | Number of blocks. |
| **/dev/null** | Number of blocks. |
| Object in **/G** | Number of blocks. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

**f_bmaxavail** Number of blocks free on the disk volume with the most space remaining.

| **For** | **Contains** |
|---|---|
| Regular file | Number of blocks. |
| Directory | Number of blocks. |
| FIFO | Number of blocks. |
| **AF_UNIX** socket | Number of blocks. |
| **/dev/null** | Number of blocks. |
| Object in **/G** | Number of blocks. |
| **/G** | 0 |
| **/G/ztnt/#pty***nn* | 0 |
| **/E** | 0 |

### Use From the Guardian Environment

The **statvfs64( )** function is one of a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. You cannot close these file numbers by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

This function provides compatibility with the System V Interface Definition, Revision 3.

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **statvfs64( )** function returns the value 0 (zero). Otherwise, it returns the value -1 and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **statvfs64( )** function sets **errno** to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix of the *path* parameter.

[EFAULT] The *buffer* or *path* parameter points to a location outside of the allocated address space of the process.

[EINTR] The function was interrupted by a signal before any data arrived.

[EIO] One of the following conditions occurred:

- The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.

- A physical I/O error has occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed. Data might have been lost during a transfer.

[ELOOP] Too many symbolic links were encountered in translating the *path* parameter.

[ENAMETOOLONG]
One of these names is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

You can call the **pathconf( )** function to obtain the applicable limits.

[ENOENT] The file referred to by the *path* parameter does not exist.

[ENOTDIR] A component of the path prefix of the *path* parameter is not a directory.

## RELATED INFORMATION

Functions: **fstat(2)**, **fstat64(2)**, **fstatvfs(2)**, **fstatvfs64(2)**, **lstat(2)**, **lstat64(2)**, **stat(2)**, **stat64(2)**.

## STANDARDS CONFORMANCE

This function is an HP extension to the XPG4 Version 2 specification.

## NAME

**symlink -** Creates a symbolic link to a file

## LIBRARY

G-series native Guardian processes: system library
G-series native OSS processes: system library
H-series native Guardian processes: implicit libraries
H-series OSS processes: implicit libraries

## SYNOPSIS

**#include <unistd.h>**

**int symlink(**
    **const char ****path1***,**
    **const char ****path2***);**

## PARAMETERS

*path1* Specifies the file for which the symbolic link must be created. The file named by the *path1* parameter does not need to exist when the link is created. The *path1* parameter can refer to a symbolic link.

*path2* Names the symbolic link to be created. An error is returned if the symbolic link named by the *path2* parameter already exists.

## DESCRIPTION

The **symlink( )** function creates a symbolic link with the name specified by the *path2* parameter, which refers to the file named by the *path1* parameter.

Like a hard link (described in the **link(2)** reference page), a symbolic link allows a file to have multiple names. The presence of a hard link guarantees the existence of a file, even after the original name has been removed; a symbolic link provides no such guarantee. Unlike hard links, a symbolic link can cross fileset boundaries.

When a component of a pathname refers to a symbolic link rather than a directory, the pathname contained in the symbolic link is resolved. If the pathname in the symbolic link starts with a slash (/) character, the symbolic link pathname is resolved relative to the root directory of the process. If the pathname in the symbolic link does not start with a slash (/) character, the symbolic link pathname is resolved relative to the directory that contains the symbolic link.

If the symbolic link is not the last component of the original pathname, the remaining components of the original pathname are appended to the contents of the link and pathname resolution continues.

The symbolic link pathname may or may not be traversed, depending on which function is being performed. Most functions traverse the link.

The functions that refer only to the symbolic link itself, rather than to the object to which the link refers, are as follows:

**link( )** An error is returned if a symbolic link is named by the *path2* parameter.

**lstat( )** If the file specified is a symbolic link, the status of the link itself is returned.

**mknod( )** An error is returned if a symbolic link is named as the *path* parameter.

**open( )** An error is returned when **O_CREAT** and **O_EXCL** are both specified and the *path* parameter specifies an existing symbolic link.

**readlink( )** This function applies only to symbolic links.

**remove( )** A symbolic link can be removed by invoking the **remove( )** function.

**rename( )** If the file to be renamed is a symbolic link, the symbolic link is renamed. If the new name refers to an existing symbolic link, the symbolic link is destroyed.

**rmdir( )** An error is returned if a symbolic link is named as the *path* parameter.

**unlink( )** A symbolic link can be removed by invoking **unlink( )**.

Execute (search) permission for the directories within a symbolic link are required to traverse the resolved pathname. Normal permission checks are made on each component of the symbolic link pathname during its resolution.

### Use on Guardian Objects

The **symlink( )** function can be used to create a symbolic link between an OSS fileset and an object in the Guardian file system (**/G**). Symbolic links cannot be created in **/G**.

### Use From the Guardian Environment

The **symlink( )** function can be used by a Guardian process when the process has been compiled using the **#define _XOPEN_SOURCE_EXTENDED 1** feature-test macro or an equivalent compiler command option.

The **symlink( )** function is one of a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file-system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

An absolute pathname that includes a symbolic link for an OSS file on a remote HP NonStop node is expanded relative to the root fileset of the remote node. For example, if *path1* is specified as **/usr/bin** and *path2* is specified as **link**, then a reference to **/E/node1/link** is expanded to **/E/node1/usr/bin** and identifies the **/usr/bin** directory on the HP NonStop node named NODE1.

## RETURN VALUES

Upon successful completion, the **symlink( )** function returns the value 0 (zero). Otherwise, the value -1 is returned and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **symlink( )** function sets **errno** to the corresponding value:

[EACCES] One of the following conditions exists:

- The requested operation requires writing in a directory with a mode that denies write permission.

- Search permission is denied on a component of *path2*.

[EEXIST] The path specified by the *path2* parameter already exists.

[EFAULT] Either the *path1* or the *path2* parameter points outside the process's allocated address space.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EIO] An input/output error occurred during a read from or write to the fileset.

[ELOOP] Too many symbolic links were found in translating *path2*.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path1* or *path2* parameter
- A component of the pathname pointed to by the *path1* or *path2* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the pathname pointed to by the *path2* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of the following conditions exists:

- A named directory does not exist.
- A specified pathname is an empty string.
- A specified pathname cannot be mapped to a valid Guardian filename.
- The *path1* or *path2* parameter specifies a file on a remote HP NonStop node but communication with the remote node has been lost.

[ENOROOT] One of the following conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node and communication with the remote name server has been lost.

[ENOSPC] The directory in which the entry for the symbolic link is being placed cannot be extended because there is no space left on the fileset containing the directory, or the new symbolic link cannot be created because there is no space left on the fileset that contains the link.

[ENOTDIR] A component of *path2* is not a directory.

[ENOTSUP] The fileset pointed to by the *path1* parameter cannot support symbolic links. This error is returned for files in **/G** and for files in D3x-series filesets.

[ENXIO] The fileset containing the client’s working directory or effective root directory is not mounted.

[EOSSNOTRUNNING]
The program attempted an operation on an object in the OSS environment while a required system process was not running.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The requested operation requires writing in a directory on a read-only fileset.

## RELATED INFORMATION

Functions: **link(2)**, **lstat(2)**, **mknod(2)**, **readlink(2)**, **remove(3)**, **rename(2)**, **rmdir(2)**, **stat(2)**, **unlink(2)**.

Commands: **ln(1)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EFSBAD], [ENOROOT], [ENOTSUP], [ENXIO], and [EOSSNOTRUNNING] can be returned.

# Section 8. System Functions (t)

This section contains reference pages for Open System Services (OSS) system function calls with names that begin with **t**. These reference pages reside in the **cat2** directory and are sorted alphabetically by U.S. English conventions in this section.

## NAME

**tdm_execve -** Executes a file with HP extensions

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

## SYNOPSIS

**#include <tdmext.h>**

[ **extern char **environ;** ]

**int tdm_execve(**
  **const char ***path***,**
  **char * const** *argv***[ ],**
  **char * const** *envp***[ ],**
  **const struct process_extension ***pe_parms***,**
  **struct process_extension_results ***pr_results***);**

## PARAMETERS

****environ** Points to an array of character pointers to environment strings. The environment strings define the OSS environment for the calling process. The **environ** array is terminated by a null pointer.

*path* Points to a null-terminated string containing a pathname that identifies the new process image file. The pathname is absolute if it starts with a slash (/) character. Otherwise, the pathname is relative and is resolved by prefixing the current working directory.

*argv***[ ]** Specifies an array of character pointers to null-terminated strings containing arguments to be passed to the main function of the new program. *argv***[0]** should point to the null-terminated string containing the filename of the new process image. The last member of this array must be a null pointer.

*envp***[ ]** Specifies an array of character pointers to null-terminated strings that describe the environment for the new process.

*pe_parms* Points to the input structure containing Guardian process attributes to be assigned to the new process. The structure must be defined locally to match the definition in the **tdmext.h** header file. The local structure must be initialized before its first use. Initialization can be done by using the **#define DEFAULT_PROCESS_EXTENSION**, as defined in the **tdmext.h** header file. The initialized values can then be modified as appropriate for the call. When this parameter contains a null pointer, the **tdm_execve( )** function assumes default Guardian attributes.

*pr_results* Points to the output structure containing optional process identification and error information. In case of error, this structure provides additional information including the PROCESS_LAUNCH_ procedure error and error detail. The structure must be defined locally to match the definition in the **tdmext.h** header file. The local structure must be initialized before its first use. Initialization can be done using the **#define DEFAULT_PROCESS_EXTENSION_RESULTS**, as defined in the **tdmext.h** header file.

See the **process_extension_results(5)** reference page for information about the content of the structure. The **tdmext.h** header file is not kept current when new error codes are defined for process creation functions. The list of **_TPC_** macros described in that reference page is not complete; for a current description of error macros and error codes, see the Guardian header file $SYSTEM.ZSPIDEF.ZGRDC or the summary of process-creation errors in the *Guardian Procedure Calls Reference Manual* (see the table entitled "Summary of Process Creation Errors").

## DESCRIPTION

The **tdm_execve( )** function replaces the current process image with a new process image. The new image is constructed from a regular executable file, called a new process image file. The new process image file is formatted as an executable text or binary file in one of the formats recognized by the **tdm_execve( )** function.

The **tdm_execve( )** function is similar to the **tdm_execvep( )** function. The main difference is the way the pathname for the process image file is resolved. **tdm_execve( )** always resolves relative pathnames by using the current working directory; see **Identifying the Process Image File**, later in this reference page. **tdm_execvep( )** sometimes uses the **PATH** environment variable to resolve pathnames.

A successful **tdm_execve( )** function call does not return, because the calling process image is overlaid by the new process image.

When a program is executed as a result of a **tdm_execve( )** call, it is entered as a function call:

**int main(**
    **int** *argc***,**
    **char** ∗*argv***[ ],**
    **char** ∗*envp***);**

Here, the *argc* parameter is the argument count, the *argv***[ ]** parameter is an array of character pointers to the arguments themselves, and the *envp* parameter is a pointer to a character array listing the environment variables. The *argv***[ ]** array is terminated by a null pointer. The null pointer is not counted in *argc*.

The arguments specified by a program using the **tdm_execve( )** function are passed on to the new process image in the corresponding arguments to the **main( )** function.

### Use From the Guardian Environment

If called from a Guardian process, the actions of this function are undefined, and **errno** is set to [ENOTOSS].

### Identifying the Process Image File

The **tdm_execve( )** function uses the *path* parameter to identify the process image file. This parameter points to the absolute pathname if the pathname starts with a slash (/) character. Otherwise, the pathname is relative and is resolved by prefixing the current working directory.

### Passing the Arguments

The *argv***[ ]** parameter is an array of character pointers to null-terminated strings. The last member of this array is a null pointer. These strings constitute the argument list available to the new process image. The value in *argv***[0]** should point to a filename that is associated with the process being started by the **tdm_execve( )** function.

### Specifying the Environment

The *envp***[ ]** parameter is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated with a null pointer.

The number of bytes available for the new process's combined argument and environment lists has a system-imposed limit. This limit, which includes the pointers and the null terminators on the strings, is available by calling the **sysconf(_SC_ARG_MAX)** function.

### Executing a Binary File

If the file specified as the new process image file is a binary executable file, the **tdm_execve( )** function loads the file directly.

### Executing a Text File

If the file specified as the new process image file is not a binary executable file, the **tdm_execve( )** function examines the file to determine whether it is an executable text file. It checks for a header line in this format:

**#!** *interpreter_name* [*optional_string*]

The **#!** notation identifies the file as an executable text file. The new process image filename is constructed from the process image filename in the *interpreter_name* string, treating it like the *path* parameter. The Guardian input and output structures pointed to by the *pe_parms* and *pr_results* parameters apply to the command interpreter as they would to any process file.

The arguments passed to the new process are modified as listed:

- *argv***[0]** is set to the name of the command interpreter.
- If the *optional_string* portion is present, *argv***[1]** is set to *optional_string*.
- The next element of *argv***[ ]** is set to the original value of *path*.
- The remaining elements of *argv***[ ]** are set to the original elements of *argv***[ ]**, starting with *argv***[1]**. The original *argv***[0]** is discarded.

The **S_ISUID** and **S_ISGID** mode bits of an executable text file are honored. Those bits are ignored for the *interpreter_name* command interpreter.

### When the File Is Invalid

If the process image file is not a valid executable object, or if the text file does not contain the header line, the **tdm_execve( )** function returns and sets **errno** to [ENOEXEC].

### Open Files

File descriptors open in the calling process image remain open in the new process image, except for those:

- Whose close-on-exec flag **FD_CLOEXEC** is set (see the **fcntl(2)** reference page)
- Opened using a Guardian function or procedure call

For a G-series TNS process image or an accelerated process image only, if the process file segment (PFS) of the new process image is smaller than the process file segment of the calling process image and if the calling process image has a large number of file descriptors open, the system might not be able to propagate all the open file descriptors to the new process image. When this situation occurs, the function call fails, and **errno** is set to the value of [EMFILE].

For those file descriptors that remain open, all attributes of the open file descriptor, including file locks, remain unchanged. All directory streams are closed.

### Open Pipes and FIFOs

A pipe or FIFO associated with an open file descriptor in the calling process remains connected in the new process. If the new process runs in a different processor than the calling process, the processor that runs the new process must also be running an OSS pipe server process.

If no OSS pipe server process is running in the new processor, the new process cannot use the pipe or FIFO; calls specifying the file descriptor for the pipe or FIFO fail with **errno** set to [EWRONGID]. The new process can only close the invalid file descriptor.

**Existing Sockets**

A socket associated with an open file descriptor in the calling process remains connected in the new process when the new process runs in the same processor as the calling process.

When the new process runs in a different processor than the calling process, the processor that runs the new process must also be running a socket transport agent process. If no socket transport agent process is running in the new processor, the new process cannot use the socket; calls specifying the file descriptor for the socket fail with **errno** set to [EWRONGID]. The new process can only close the invalid file descriptor.

**Shared Memory**

Any attached shared memory segments are detached from the new process by a successful call to the **tdm_execve( )** function. See the **shmat(2)** reference page for additional information about shared memory segment use.

**Semaphores**

Semaphore set IDs attached to a calling process are also attached to the new process if the new process executes in the same processor as the calling process. The new process also inherits the adjust-on-exit (**semadj**) values of the calling process if both processes are in the same processor.

A semaphore set cannot be shared when a **semadj** value exists for the calling process and the new process is created in a different processor. When that condition exists, a call to the **tdm_execve( )** function fails, and **errno** is set to [EHLDSEM].

See the **semget(2)** reference page for additional information about semaphore use.

**Signals**

Signals set to:

- The default action (**SIG_DFL**) in the calling process image are set to the default action in the new process image.
- Be ignored (**SIG_IGN**) by the calling process image are set to be ignored by the new process image.
- Cause abnormal termination (**SIG_ABORT**) in the calling process image are set to that action in the new process image.
- Cause entry into the debugger (**SIG_DEBUG**) in the calling process image are set to that action in the new process image.
- Be caught by the calling process image are set to the default action in the new process image.

See the **signal(4)** reference page either online or in the *Open System Services System Calls Reference Manual*.

**User ID and Group ID**

If the set-user-ID mode bit (**S_ISUID**) of the new process image file is set (see the **chmod(2)** reference page), the effective user ID of the new process image is set to the user ID of the owner of the new process image file. Similarly, if the set-group-ID mode bit (**S_ISGID**) of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved-set user ID and the saved-set group ID) for use by the **setuid( )** function.

### OSS Attributes

These OSS attributes of the calling process image are unchanged after successful completion of the **tdm_execve( )** function:

- OSS process ID (PID)
- Parent OSS process ID
- Process group ID
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- The time left until an alarm clock signal is posted (see the **alarm(3)** reference page)
- Current working directory
- Root directory
- File mode creation mask (see the **umask(2)** reference page)
- Process signal mask (see the **sigprocmask(2)** reference page)
- Pending signals (see the **sigpending(2)** reference page)
- The **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** fields of the **tms** structure
- File size limit (see the **ulimit(2)** reference page)

Upon successful completion, the **tdm_execve( )** function marks the **st_atime** field of the file for update.

The POSIX.1 standard does not specify the effect on the **st_atime** field when the **tdm_execve( )** function call fails but does find the file. Neither does the HP implementation guarantee the outcome. Under these circumstances, this field should not be used for further processing.

### Default Guardian Attributes

If the *pe_parms* parameter contains a null pointer, the newly created OSS process retains all these default Guardian attributes of the process that calls the **tdm_execve( )** function:

- Priority
- Processor on which the process executes
- Home terminal
- Job ID
- DEFINE mode switch
- Process access ID (PAID), unless the **S_ISUID** mode bit of the new process image file is set

- Security group list
- Job ancestor or GMOM
- Unread system message index (PCBMCNT)

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Outstanding incoming and outgoing message limits

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Login, remote login, and saveabend flags
- File creation mask

If the *pe_parms* parameter contains a null pointer, the default Guardian attributes of the new process that differ from those of the calling process are:

- The program file is the file specified in the **tdm_execve( )** call.
- The library file is specified in the program file.
- The process name for the new process is system-generated if the RUNNAMED option is set in the program file. Otherwise the process is unnamed.
- The size of the data segment of the new process is set in the program file.
- The remote login flag (PCBREMID) is set to zero (off) if the program file has its **S_ISUID** mode bit set. Otherwise, the remote login flag is set the same as for the caller.
- The size of the extended data segment of the new process is set in the program file.
- The DEFINEs inherited by the new process depend on the setting of DEFINE mode in the caller. If DEFINE mode in the caller is ON, all the caller's DEFINEs are inherited. If DEFINE mode is OFF, no DEFINEs are inherited.
- The new process does not inherit the extended swap file (if any) of the calling process. For a G-series TNS process or an accelerated process, the extended data segment is managed by the Kernel Managed Storage Facility (KMSF) unless an extended swap file is specified in the **pe_extswap_file_name** field of the **process_extension** structure described elsewhere in this reference page.
- The process identification number (PIN) of the new process is unrelated to that of the calling process. Usually, the PIN of the new process is unrestricted. However, the PIN can be restricted to the range 0 through 254 under the following conditions:

  — The HIGHPIN flag is not set in, or is absent from, the program file or any library file.

  — **_TPC_HIGHPIN_OFF** is specified in the **pe_create_options** field of the **process_extension** structure, described elsewhere in this reference page.

  — The restriction is inherited. See **_TPC_IGNORE_FORCEPIN_ATTR** in the **pe_create_options** field of the **process_extension** structure, described elsewhere in this reference page, for more information about controlling inheritance.

- The creator access ID (CAID) is set to the process access ID (PAID) of the calling process.
- The PAID depends on whether the **S_ISUID** mode bit of the process image file is set. If that bit is set, the PAID is based on the file owner ID. If not, the PAID is the same as for the caller. (The **S_ISUID** mode bit of the image file has no effect on the security group list.)
- The MOM field for the new process depends on whether the calling process is named. If it is named, the MOM field of the new process is set to the caller's ANCESTOR field. Otherwise, the MOM field of the new process is set to the caller's MOM field.
- System debugger selection for the new process is based on INSPECT mode.
- Code breakpoints and memory breakpoints are not inherited.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

**Setting Guardian Attributes**

The input structure pointed to by the *pe_parms* parameter permits the setting of Guardian attributes for the new process.

First, the input structure must be initialized to the default values (see **Default Guardian Attributes**, earlier in this reference page) using the **#define DEFAULT_PROCESS_EXTENSION**. After the structure is initialized, the values can be set using literals that are defined in the **tdmext.h** header file.

If any optional parameter specified in the structure pointed to by *pe_parms* is not passed, the new process assumes the corresponding default value.

The input structure is defined in the **tdmext.h** header file. This structure contains fields that can vary from release version update (RVU) to RVU, including reserved and filler fields.

In the current RVU, these fields are meaningful:

```
#if defined (__LP64) || defined (_PROCEX32_64BIT)
    typedef struct process_extension {
            short    pe_ver;
            short    pe_len;
            int      pe_pfs_size;
            long long    pe_mainstack_max;
            long long    pe_heap_max;
            long long    pe_space_guarantee;
            char _ptr64  *pe_library_name;
            char _ptr64  *pe_swap_file_name;
            char _ptr64  *pe_extswap_file_name;
            char _ptr64  *pe_process_name;
            char _ptr64  *pe_hometerm;
            char _ptr64  *pe_defines;
            short    pe_defines_len;
            short    pe_priority;
            short    pe_cpu;
            short    pe_memory_pages;
            short    pe_jobid;
            short    pe_name_options;
            short    pe_create_options;
            short    pe_debug_options;
            short    pe_OSS_options;
            char     filler_1[6];
    } process_extension_def;
#else  /* !defined (__LP64) && !defined (_PROCEX32_64BIT) */
    typedef struct process_extension {
            long     pe_len;
            char     *pe_library_name;
            char     *pe_swap_file_name;
            char     *pe_extswap_file_name;
            short    pe_priority;
            short    pe_cpu;
            short    pe_name_options;
            char     filler_1[2];
            char     *pe_process_name;
            char     *pe_hometerm;
            short    pe_memory_pages;
            short    pe_jobid;
            short    pe_create_options;
            char     filler_2[2];
            char     *pe_defines;
            short    pe_defines_len;
            short    pe_debug_options;
            long     pe_pfs_size;
            short    pe_OSS_options;
            char     filler_3[2];
            long     pe_mainstack_max;
            long     pe_heap_max;
            long     pe_space_guarantee;
    } process_extension_def;
```

**#endif /* !defined (__LP64) && !defined (_PROCEX32_64BIT) */**

When an application is compiled in 64-bit compile mode or compiled using the **#define _PROCEX32_64BIT 1** feature test macro or an equivalent compiler command option, the application will use the version of the **process_extension** structure that contains 64-bit data types. The **_PROCEX32_64BIT** flag is only required if a 32-bit process must specify larger 64-bit values for **pe_mainstack_max**, **pe_heap_max**, and **pe_space_guaranter**. These larger data types are optional when creating a 64-bit process.

**Note:** The input structure supports two versions: one that contains 64-bit data types and one that contains 32-bit data types. Because the order in which the fields appear in this structure varies significantly based on the version in use, the field definitions below are defined alphabetically instead of sequentially.

The input structure passes this information:

**pe_cpu** Specifies the processor on which the new process will execute. The OSS process ID (PID) of the process remains unchanged. This field is used to distribute system load.

**pe_create_options**
Specifies process creation options as:

**_TPC_BOTH_DEFINES**
Propagates the current DEFINEs and the DEFINEs indicated in the input structure.

**_TPC_ENABLE_DEFINES**
Enables DEFINEs when set if **_TPC_OVERRIDE_DEFMODE** is also set. Disables DEFINEs when not set.

**_TPC_HIGHPIN_OFF**
Restricts the new process to a PIN in the range 0 through 254. This restriction is rarely useful for an OSS process; it allows obsolescent Guardian interfaces to interact with the process.

By default, this restriction is inherited by any child or successor process. The default can be overridden by using the **_TPC_IGNORE_FORCEPIN_ATTR** field.

**_TPC_IGNORE_FORCEPIN_ATTR**
Ignores the **_TPC_HIGHPIN_OFF** restriction specified for or inherited by the caller or parent process. When **_TPC_IGNORE_FORCEPIN_ATTR** is specified, the resulting process has a restricted PIN only if **_TPC_HIGHPIN_OFF** is also specified or if the object file for the program or a user library lacks the HIGHPIN attribute.

**_TPC_OVERRIDE_DEFMODE**
Specifies that the DEFINE mode of the new process is to be set according to the **_TPC_ENABLE_DEFINES** option rather than to the caller's current DEFINE mode.

**_TPC_PROCESS_DEFINES_ONLY**
Propagates only the current set of DEFINEs.

**_TPC_SUPPLIED_DEFINES_ONLY**
Propagates only the DEFINEs indicated by the **pe_defines** field.

**pe_debug_options**
Provides control over the selection between the default and symbolic debuggers and over the creation of the saveabend file. A saveabend file can be examined by using the symbolic debugger to determine the cause of the abnormal termination. In addition, you can use this option to force the new process to enter the default debugger before executing. Possible options are:

**_TPC_CODEFILE_INSPECT_SAVEABEND**
Uses the saveabend and INSPECT mode flags in the program file.

**_TPC_DEBUG_NOSAVE**
Uses the default debugger but does not create a saveabend file.

**_TPC_DEBUG_SAVEABEND**
Uses the default debugger and creates a saveabend file.

**_TPC_ENTER_DEBUG**
Starts the new process in the default debugging utility.

**_TPC_INSPECT_NOSAVE**
Uses the symbolic debugger but does not create a saveabend file.

**_TPC_INSPECT_SAVEABEND**
Uses the symbolic debugger and creates a saveabend file.

**pe_defines** Points to a specified saved set of DEFINEs created by using the Guardian DEFINESAVE procedure. These DEFINEs are propagated to the new process if either **_TPC_SUPPLIED_DEFINES_ONLY** or **_TPC_BOTH_DEFINES** is specified in the **pe_create_options** field.

**Note:** This string is not null-terminated.

**pe_defines_len**
Specifies the length of the string in the **pe_defines** field.

**pe_extswap_file_name**
Points to a null-terminated string specifying the name of a disk file in the Guardian file system to be used as the swap file for the extended data segment. For example, if the Guardian filename is $A.B.D, the name used is **/G/a/b/d**.

This file cannot have the same name as that of a file used in a preceding call to the **tdm_fork( )** function.

This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is checked for validity but otherwise ignored.

By default, the new process uses KMSF to manage its extended swap segment. HP recommends using the default.

**pe_heap_max** Specifies the maximum size of the heap in bytes for the new process if it is a native process.

See the *C/C++ Programmer's Guide* description of the **HEAP** pragma for guidance on the use of nonzero values for this field.

If a value is specified for this field for G-series TNS or accelerated object files, the specified value is ignored.

**pe_hometerm** Points to the null-terminated name in the Guardian file system for the home terminal. For example, if the Guardian name is $ztnt.#xyz, the name used is **/G/ztnt/#xyz**.

**pe_jobid** Specifies the job ID of the new process.

**pe_len** Specifies the size of the structure in bytes. This value is set by **#define DEFAULT_PROCESS_EXTENSION** and should not be changed.

**pe_library_name**

Points to the name of the user library to be bound to the new process. The string that is pointed to is null-terminated and in OSS name format. If the pointer points to a zero-length string (a NULL character), the new process runs with no user library. An equivalent call to the Guardian PROCESS_LAUNCH_ procedure does this by setting the library filename length to -1.

This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is ignored.

**pe_mainstack_max**

Specifies the maximum size of the main stack in bytes for the new process.

If the calling process specifies a value, the value must be less than 32 MB. If the calling process does not specify a value or specifies a 0 (zero) value, the value specified in the object file of the new process is used. If no value is specified in the object file, the default value of 1 MB (for TNS/R systems) or 2 MB (for TNS/E systems) is used.

**pe_memory_pages**

Specifies the size of the data stack in 2 KB units. This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is checked for validity but otherwise ignored.

**pe_name_options**

Specifies process naming as:

**_TPC_GENERATE_NAME**

The system generates the name.

**_TPC_NAME_SUPPLIED**

The process name is indicated by the **pe_process_name** field.

**_TPC_NO_NAME**

The new process is unnamed.

**pe_OSS_options**

Specifies OSS options. No special action on signals is the default and only current OSS option.

**pe_pfs_size** Specifies the size of the PFS for the new process (this field is ignored).

**pe_priority** Specifies the priority of the new process.

**pe_process_name**
Points to the null-terminated Guardian process name if **_TPC_NAME_SUPPLIED** is specified in the **pe_name_options** field. For example, if the Guardian process name is $DELM, the name used is **/G/delm**.

**pe_space_guarantee**
Specifies the minimum available swap space to guarantee for the new process.

If the calling process specifies a value, the value must be less than or equal to a multiple of the page size of the processor in which the new process will run. Values less than a multiple of the page size are rounded up to the next multiple of the page size. If the calling process does not specify a value or specifies a 0 (zero) value, the value specified in the native object file of the new process is used. If no value is specified in the native object file, the default value of 0 (zero) is used, and enough swap space is guaranteed to launch the process.

If the new process requires a guarantee of available swap space and the system cannot guarantee the required amount, the function call fails, and **errno** is set to the value of [EAGAIN].

If a value is specified for this field for G-series TNS or accelerated object files, the specified value is used for the main stack of the new process.

**pe_swap_file_name**
Points to a null-terminated string specifying the name of a file in the Guardian file system to be used as the swap file for the stack segment. For example, if the Guardian filename is $A.B.C, the name used is **/G/a/b/c**.

This file cannot have the same name as that of a file used in a preceding call to the **tdm_fork( )** function.

This field is not used in the current RVU of Open System Services. It exists for compatibility with older RVUs. Any specified value is checked for validity but otherwise ignored.

**pe_ver** Specifies the version of the **process_extension** structure. This value is set by **#define DEFAULT_PROCESS_EXTENSION** and should not be changed.

The MOM and ANCESTOR fields in the new process differ from those of a process created in the Guardian environment if the new process is named (the **pe_name_options** field is set to **_TPC_NAME_SUPPLIED** or **_TPC_GENERATE_NAME**). If the calling process is unnamed, then the ANCESTOR field for the new process is set to the caller's MOM field, and the MOM field of the new process is null. If the calling process is named, the ANCESTOR field of the new process is set to the ANCESTOR field of the calling process, and the MOM field of the new process is null.

The MOM and ANCESTOR fields for the new process are the same as for a process created in the Guardian environment if the new process is unnamed (the **pe_name_options** field is set to **_TPC_NO_NAME**). If the caller is unnamed, the MOM field for the new process is set to the MOM field of the caller. If the caller is named, the MOM field for the new process is set to the ANCESTOR field of the calling process.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Output Structure Information

If the *pr_results* parameter does not contain a null pointer, it points to an output structure defined in the **tdmext.h** header file. This structure can contain fields that vary from RVU to RVU, including reserved and filler fields.

First, the output structure must be initialized by using the **#define DEFAULT_PROCESS_EXTENSION_RESULTS**. This initialization sets the value of the **pr_len** field to the correct value for the current RVU. The value of the **pr_len** field should not be modified after being set by **#define DEFAULT_PROCESS_EXTENSION_RESULTS**.

The **process_extension_results** output structure is described in the **process_extension_results(5)** reference page.

## RETURN VALUES

If the **tdm_execve( )** function returns to the calling process image, an error has occurred; the return value is -1, and **errno** is set to indicate the error. If the *pr_results* parameter does not contain a null pointer, the structure it points to returns additional error information, including the PROCESS_LAUNCH_ error and error detail.

## ERRORS

If any of the following conditions occurs, the **tdm_execve( )** function sets **errno** to the corresponding value, file descriptors marked close-on-exec are not closed, signals set to be caught are not set to the default action, and none of these are changed:

- The *argv*[ ] array of pointers
- The *envp*[ ] array of pointers
- The elements pointed to by these arrays
- The value of the global variable **environ**
- The pointers contained within the global variable **environ**
- The elements pointed to by **environ** pointers
- The effective user ID of the current process
- The effective group ID of the current process

[E2BIG] The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit. The limit can be obtained by calling the **sysconf(_SC_ARG_MAX)** function.

[EACCES] One of these conditions exists:

- Search permission is denied for the directory components of the pathname prefix to the process image file.
- The new process image file, any library file, or script file denies execution permission.
- The new process image file is not a regular file.

[EAGAIN] System resources such as disk space, process control block (PCB) space, MAPPOOL space, stack space, or PFS space are temporarily inadequate.

[EFAULT] An address for a parameter in the **process_extension** structure pointed to by *pe_parms* is out of allowable bounds. The Guardian PROCESS_LAUNCH_ error and error detail information is returned in the structure pointed to by the *pr_results* parameter, unless *pr_results* contains a null pointer.

[EHLDSEM] The process tried to create a new process in a different processor while having at least one **semadj** value.

[EINVAL] One of these conditions exists:

- An invalid parameter value was supplied in the **process_extension** structure pointed to by *pe_parms*. The Guardian PROCESS_LAUNCH_ error and error detail information is returned in the structure pointed to by the *pr_results* parameter, unless *pr_results* contains a null pointer.
- The new process image file is a binary executable file with invalid attributes.

[EIO] Some physical input or output error has occurred. Either a file cannot be opened because of an input or output error, or data has been lost during an input or output transfer. This value is used for errors on the object file of a loaded program or library, or during data transfer with a Guardian environment home terminal.

For systems running J06.07 and later J-series RVUs or H06.18 or later H-series RVUs, this error can also occur when the OSS file system is out of memory and one or more open files cannot be propagated from the parent process to the child process. In this case, if you are running a program from the shell with the shell reporting any errors, you might see an error like this:

/bin/-sh: /bin/ps: tdm_execve(): failed with unexpected error pr_errno=(4005) pr_TPCerror=(110) pr_TPCdetail=(36)

where:

- **pr_errno** is the [EIO] error
- **pr_TPCerror** is the Guardian PROCESS_LAUNCH_ or PROCESS_CREATE_ error.

[ELOOP] Too many symbolic links were encountered in pathname resolution.

[EMFILE] The maximum number of files are open. The process attempted to open more than the maximum number of file descriptors allowed for the process. The process file segment (PFS) of the new process might be smaller than that of the calling process.

[ENAMETOOLONG]
One of these is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the pathname pointed to by the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOCPU] The selected processor does not exist, or the selected processor is down or otherwise unavailable for process creation.

[ENODEV] The system cannot find the file system containing the process image file.

[ENOENT] One of these conditions exists:

- One or more components of the new process image file's pathname do not exist.
- The *path* parameter points to an empty string.

[ENOEXEC] The new process image file has the appropriate access permissions and is in the OSS name space, but it is neither in the correct binary executable format nor a valid executable text file.

[ENOMEM] Required resources are not available. Subsequent calls to the same function will not succeed for the same reason.

Possible causes of this error include insufficient primary memory (stack, globals, or heap) for the new process.

[ENOTDIR] A component of the path prefix of the new process image file is not a directory.

[ENOTOSS] The calling process is not an OSS process. The **tdm_execve( )** function cannot be called from the Guardian environment.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[ETXTBSY] The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.

[EUNKNOWN]
Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

## RELATED INFORMATION

Commands: **eld(1)**, **ld(1)**, **nld(1)**.

Functions: **alarm(3)**, **chmod(2)**, **exec(2)**, **_exit(2)**, **exit(3)**, **fcntl(2)**, **fork(2)**, **getenv(3)**, **putenv(3)**, **semget(2)**, **shmat(2)**, **sigaction(2)**, **system(3)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **times(3)**, **ulimit(2)**, **umask(2)**.

Files: **signal(4)**.

Miscellaneous: **environ(5)**, **process_extension_results(5)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification.

## NAME

**tdm_execvep -** Executes a file with HP extensions

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

## SYNOPSIS

```
#include <tdmext.h>

[ extern char **environ; ]

int tdm_execvep(
        const char *file,
        char * const argv[ ],
        char * const envp[ ],
        const struct process_extension *pe_parms,
        struct process_extension_results *pr_results);
```

## PARAMETERS

****environ**
Points to an array of character pointers to environment strings. The environment strings define the OSS environment for the calling process. The **environ** array is terminated by a null pointer.

*file*
Points to a pathname that identifies the new process image file. If the pathname starts with a slash (/) character, it is the absolute pathname. If this pathname does not start with a slash but does contain a slash, the pathname resolves relative to the current working directory. Otherwise, the pathname contains no slash, and the system searches the directories listed in the **PATH** environment variable for the file and prefixes the directory in which the file is found.

*argv***[ ]**
Specifies an array of character pointers to null-terminated strings containing arguments to be passed to the main function of the new program. *argv***[0]** should point to the null-terminated string containing the filename of the new process image. The last member of this array must be a null pointer.

*envp***[ ]**
Specifies an array of character pointers to null-terminated strings that describe the environment for the new process.

*pe_parms*
Points to the input structure containing Guardian process attributes to be assigned to the new process. The structure must be defined locally to match the definition in the **tdmext.h** header file. The local structure must be initialized before its first use. Initialization can be done using the **#define DEFAULT_PROCESS_EXTENSION**, as defined in the **tdmext.h** header file. The initialized values can then be modified as appropriate for the call. When this parameter contains a null pointer, the **tdm_execvep( )** function assumes default Guardian attributes.

*pr_results*
Points to the output structure containing optional process identification and error information. In case of error, this structure provides additional information, including the PROCESS_LAUNCH_ procedure error and error detail. The structure must be defined locally to match the definition in the **tdmext.h** header file. The local structure must be initialized before its first use. Initialization can be done using the **#define DEFAULT_PROCESS_EXTENSION_RESULTS**, as defined in the **tdmext.h** header file.

See the **process_extension_results(5)** reference page for information about the content of the structure. The **tdmext.h** header file is not kept current when new error codes are defined for process creation functions. The list of **_TPC_** macros described in that reference page is not complete; for a current description of error macros and error codes, see the Guardian header file $SYSTEM.ZSPIDEF.ZGRDC or the summary of process-creation errors in the *Guardian Procedure Calls Reference Manual* (see the table entitled "Summary of Process Creation Errors").

## DESCRIPTION

The **tdm_execvep( )** function replaces the current process image with a new process image. The new image is constructed from a regular executable file, called a new process image file. The new process image file is formatted as an executable text or binary file in one of the formats recognized by the **tdm_execvep( )** function.

The **tdm_execvep( )** function is similar to the **tdm_execve( )** function. The main difference is in the way the pathname for the process image file is resolved. **tdm_execve( )** always resolves relative pathnames by prefixing the current working directory. **tdm_execvep( )** sometimes uses the **PATH** environment variable; see **Identifying the Process Image File**, later in this reference page.

A successful **tdm_execvep( )** function call does not return, because the calling process image is overlaid by the new process image.

When a program is executed as a result of a **tdm_execvep( )** call, it is entered as a function call:

**int main(**
    **int** *argc***,**
    **char** ***argv***[ ],**
    **char** ***envp***);**

Here, the *argc* parameter is the argument count, the *argv***[ ]** parameter is an array of character pointers to the arguments themselves, and the *envp* parameter is a pointer to a character array listing the environment variables. The *argv***[ ]** array is terminated by a null pointer. The null pointer is not counted in *argc*.

The arguments specified by a program using the **tdm_execvep( )** function are passed on to the new process image in the corresponding arguments to the **main( )** function.

### Use From the Guardian Environment

If called from a Guardian process, the actions of this function are undefined, and **errno** is set to [ENOTOSS].

### Identifying the Process Image File

The **tdm_execvep( )** function uses the *file* parameter to identify the process image file. If the pathname specified as the *file* parameter starts with a slash (/) character, it is the absolute pathname. If the pathname does not start with a slash but contains a slash, the pathname is resolved relative to the current working directory. Otherwise, the pathname does not contain a slash, and the system searches the directories listed in the **PATH** environment variable for the file and prefixes the directory in which the file is found.

### Passing the Arguments

The *argv***[ ]** parameter is an array of character pointers to null-terminated strings. The last member of this array is a null pointer. These strings constitute the argument list available to the new process image. The value in *argv***[0]** should point to a filename that is associated with the process being started by the **tdm_execvep( )** function.

### Specifying the Environment

The *envp***[ ]** parameter is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated with a null pointer.

The number of bytes available for the new process's combined argument and environment lists has a system-imposed limit. This limit, which includes the pointers and the null terminators on the strings, is available by calling the **sysconf(_SC_ARG_MAX)** function.

### Executing a Binary File

If the file specified as the new process image file is a binary executable file, the **tdm_execvep( )** function loads the file directly.

### Executing a Text File

If the file specified as the new process image file is not a binary executable file, the **tdm_execvep( )** function examines the file to determine whether it is an executable text file. It checks for a header line in this format:

**#!** *interpreter_name* [*optional_string*]

The **#!** notation identifies the file as an executable text file. The new process image filename is constructed from the process image filename in the *interpreter_name* string, treating it like the *file* parameter. The Guardian input and output structures pointed to by the *pe_parms* and *pr_results* parameters apply to the command interpreter as they would to any process file.

The arguments passed to the new process are modified as listed:

- *argv***[0]** is set to the name of the command interpreter.
- If the *optional_string* portion is present, *argv***[1]** is set to *optional_string*.
- The next element of *argv***[ ]** is set to the original value of *file*.
- The remaining elements of *argv***[ ]** are set to the original elements of *argv***[ ]**, starting with *argv***[1]**. The original *argv***[0]** is discarded.

The **S_ISUID** and **S_ISGID** mode bits of an executable text file are honored. Those bits are ignored for the *interpreter_name* command interpreter.

### When the File Is Invalid

If the process image file is not a valid executable object, or if the text file does not contain the header line, the **tdm_execvep( )** function invokes the *interpreter_name* command interpreter as the new process image and passes these arguments to it:

- *argv***[0]** is set to the string **sh**.
- *argv***[1]** is set to the original value of the *file* parameter.
- The remaining elements of *argv***[ ]** are set to the original elements of *argv***[ ]**, starting with *argv***[1]**.
- The original *argv***[0]** is discarded.

**Open Files**

File descriptors open in the calling process image remain open in the new process image, except for those:

- Whose close-on-exec flag **FD_CLOEXEC** is set (see the **fcntl(2)** reference page)
- Opened using a Guardian function or procedure call

For a G-series TNS process image or an accelerated processs image only, if the process file segment (PFS) of the new process image is smaller than the process file segment of the calling process image and if the calling process image has a large number of file descriptors open, the system might not be able to propagate all the open file descriptors to the new process image. When this situation occurs, the function call fails, and **errno** is set to the value of [EMFILE].

For those file descriptors that remain open, all attributes of the open file descriptor, including file locks, remain unchanged. All directory streams are closed.

**Open Pipes and FIFOs**

A pipe or FIFO associated with an open file descriptor in the calling process remains connected in the new process. If the new process runs in a different processor than the calling process, the processor that runs the new process must also be running an OSS pipe server process.

If no OSS pipe server process is running in the new processor, the new process cannot use the pipe or FIFO; calls specifying the file descriptor for the pipe or FIFO fail with **errno** set to [EWRONGID]. The new process can only close the invalid file descriptor.

**Existing Sockets**

A socket associated with an open file descriptor in the calling process remains connected in the new process when the new process runs in the same processor as the calling process.

When the new process runs in a different processor than the calling process, the processor that runs the new process must also be running a socket transport agent process. If no socket transport agent process is running in the new processor, the new process cannot use the socket; calls specifying the file descriptor for the socket fail with **errno** set to [EWRONGID]. The new process can only close the invalid file descriptor.

**Shared Memory**

Any attached shared memory segments are detached from the new process by a successful call to the **tdm_execvep( )** function. See the **shmat(2)** reference page for additional information about shared memory segment use.

**Semaphores**

Semaphore set IDs attached to the calling process are also attached to the new process if the new process executes in the same processor as the calling process. The new process also inherits the adjust-on-exit (**semadj**) values of the calling process if both processes are in the same processor.

A semaphore set cannot be shared when a **semadj** value exists for the calling process and the new process is created in a different processor. When that condition exists, a call to the **tdm_execvep( )** function fails and **errno** is set to [EHLDSEM].

See the **semget(2)** reference page for additional information about semaphore use.

**Signals**

Signals set to:

- The default action (**SIG_DFL**) in the calling process image are set to the default action in the new process image.

- Be ignored (**SIG_IGN**) by the calling process image are set to be ignored by the new process image.
- Cause abnormal termination (**SIG_ABORT**) in the calling process image are set to that action in the new process image.
- Cause entry into the debugger (**SIG_DEBUG**) in the calling process image are set to that action in the new process image.
- Be caught by the calling process image are set to the default action in the new process image.

See the **signal(4)** reference page either online or in the *Open System Services System Calls Reference Manual*.

**User ID and Group ID**

If the set-user-ID mode bit (**S_ISUID**) of the new process image file is set (see the **chmod(2)** reference page), the effective user ID of the new process image is set to the user ID of the owner of the new process image file. Similarly, if the set-group-ID mode bit (**S_ISGID**) of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved-set user ID and the saved-set group ID) for use by the **setuid( )** function.

**OSS Attributes**

These OSS attributes of the calling process image are unchanged after successful completion of the **tdm_execvep( )** function:

- OSS process ID (PID)
- Parent OSS process ID
- Process group ID
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- The time left until an alarm clock signal is posted (see the **alarm(3)** reference page)
- Current working directory
- Root directory
- File mode creation mask (see the **umask(2)** reference page)
- Process signal mask (see the **sigprocmask(2)** reference page)
- Pending signals (see the **sigpending(2)** reference page)
- The **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** fields of the **tms** structure

- File size limit (see the **ulimit(2)** reference page)

Upon successful completion, the **tdm_execvep( )** function marks the **st_atime** field of the file for update.

The POSIX.1 standard does not specify the effect on the **st_atime** field when the **tdm_execvep( )** function call fails but does find the file. Neither does the HP implementation guarantee the outcome. Under these circumstances, this field should not be used for further processing.

**Default Guardian Attributes**

If the *pe_parms* parameter contains a null pointer, the newly created OSS process retains all these default Guardian attributes of the process that calls the **tdm_execvep( )** function:

- Priority
- Processor on which the process executes
- Home terminal
- Job ID
- DEFINE mode switch
- Process access ID (PAID), unless the **S_ISUID** mode bit of the new process image file is set
- Security group list
- Job ancestor or GMOM
- Unread system message index (PCBMCNT)

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Outstanding incoming and outgoing message limits

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Login, remote login, and saveabend flags
- File creation mask

If the *pe_parms* parameter contains a null pointer, the default Guardian attributes of the new process that differ from those of the calling process are:

- No segments created or shared using Guardian system procedures such as SEGMENT_ALLOCATE_ are inherited.
- The program file is the file specified in the **tdm_execvep( )** call.
- The library file is specified in the program file.
- The child process does not inherit the parent process extended swap file (if any). For a G-series TNS process or an accelerated process, the extended data segment is managed by the Kernel Managed Storage Facility (KMSF) unless an extended swap file is specified in the **pe_extswap_file_name** field of the **process_extension** structure described elsewhere in this reference page.

- The process name for the new process is system-generated if the RUNNAMED option is set in the program file. Otherwise the process is unnamed.
- The size of the data segment of the new process is set in the program file.
- The remote login flag (PCBREMID) is set to zero (off) if the program file has its **S_ISUID** mode bit set. Otherwise, the remote login flag is set the same as for the caller.
- The size of the extended data segment of the new process is set in the program file.
- The DEFINEs inherited by the new process depend on the setting of DEFINE mode in the caller. If DEFINE mode in the caller is ON, all the caller's DEFINEs are inherited. If DEFINE mode is OFF, no DEFINEs are inherited.
- The process identification number (PIN) of the new process is unrelated to that of the calling process. Usually, the PIN of the new process is unrestricted. However, the PIN can be restricted to the range 0 through 254 under the following conditions:
  - — The HIGHPIN flag is not set in, or is absent from, the program file or any library file.
  - — **_TPC_HIGHPIN_OFF** is specified in the **pe_create_options** field of the **process_extension** structure.
  - — The restriction is inherited. See the description of **_TPC_IGNORE_FORCEPIN_ATTR** in the **pe_create_options** field of the **process_extension** structure for more information about controlling inheritance.
- The creator access ID (CAID) is set to the process access ID (PAID) of the calling process.
- The PAID depends on whether the **S_ISUID** mode bit of the process image file is set. If that bit is set, the PAID is based on the file owner ID. If not, the PAID is the same as for the caller. (The **S_ISUID** mode bit of the image file has no effect on the security group list.)
- The MOM field for the new process depends on whether the calling process is named. If it is named, the MOM field of the new process is set to the caller's ANCESTOR field. Otherwise, the MOM field of the new process is set to the caller's MOM field.
- System debugger selection for the new process is based on INSPECT mode.
- Code breakpoints and memory breakpoints are not inherited.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Setting Guardian Attributes

The input structure pointed to by the *pe_parms* parameter permits the setting of Guardian attributes for the new process.

First, the input structure must be initialized to the default values (see **Default Guardian Attributes**, earlier in this reference page) using the **#define DEFAULT_PROCESS_EXTENSION**. After the data structure is initialized, the values can be set using literals that are defined in the **tdmext.h** header file.

If any optional parameter specified in the structure pointed to by *pe_parms* is not passed, the new process assumes the corresponding default value.

The input structure is defined in the **tdmext.h** header file. This structure can contain fields that vary from release version update (RVU) to RVU, including reserved and filler fields.

In the current release version update (RVU), these fields are meaningful:

```
#if defined (__LP64) || defined (_PROCEX32_64BIT)
   typedef struct process_extension {
            short    pe_ver;
            short    pe_len;
            int      pe_pfs_size;
            long long    pe_mainstack_max;
            long long    pe_heap_max;
            long long    pe_space_guarantee;
            char _ptr64  *pe_library_name;
            char _ptr64  *pe_swap_file_name;
            char _ptr64  *pe_extswap_file_name;
            char _ptr64  *pe_process_name;
            char _ptr64  *pe_hometerm;
            char _ptr64  *pe_defines;
            short    pe_defines_len;
            short    pe_priority;
            short    pe_cpu;
            short    pe_memory_pages;
            short    pe_jobid;
            short    pe_name_options;
            short    pe_create_options;
            short    pe_debug_options;
            short    pe_OSS_options;
            char     filler_1[6];
   } process_extension_def;
#else  /* !defined (__LP64) && !defined (_PROCEX32_64BIT) */
   typedef struct process_extension {
            long     pe_len;
            char     *pe_library_name;
            char     *pe_swap_file_name;
            char     *pe_extswap_file_name;
            short    pe_priority;
            short    pe_cpu;
            short    pe_name_options;
            char     filler_1[2];
            char     *pe_process_name;
            char     *pe_hometerm;
            short    pe_memory_pages;
            short    pe_jobid;
            short    pe_create_options;
            char     filler_2[2];
            char     *pe_defines;
            short    pe_defines_len;
            short    pe_debug_options;
            long     pe_pfs_size;
            short    pe_OSS_options;
            char     filler_3[2];
            long     pe_mainstack_max;
```

```
   long    pe_heap_max;
           long    pe_space_guarantee;
   } process_extension_def;
#endif /* !defined (__LP64) && !defined (_PROCEX32_64BIT) */
```

When an application is compiled in 64-bit compile mode or compiled using the **#define _PROCEX32_64BIT 1** feature test macro or an equivalent compiler command option, the application will use the version of the **process_extension** structure that contains 64-bit data types. The **_PROCEX32_64BIT** flag is only required if a 32-bit process must specify larger 64-bit values for **pe_mainstack_max**, **pe_heap_max**, and **pe_space_guaranter**. These larger data types are optional when creating a 64-bit process.

**Note:** The input structure supports two versions: one that contains 64-bit data types and one that contains 32-bit data types. Because the order in which the fields appear in this structure varies significantly based on the version in use, the field definitions below are defined alphabetically instead of sequentially.

The input structure passes this information:

**pe_cpu** Specifies the processor on which the new process will execute. The OSS process ID (PID) of the process remains unchanged. This field is used to distribute system load.

**pe_create_options**
Specifies process creation options as:

**_TPC_BOTH_DEFINES**
Propagates the current DEFINEs and the DEFINEs indicated in the input structure.

**_TPC_ENABLE_DEFINES**
Enables DEFINEs when set if **_TPC_OVERRIDE_DEFMODE** is also set. Disables DEFINEs when not set.

**_TPC_HIGHPIN_OFF**
Restricts the new process to a PIN in the range 0 through 254. This restriction is rarely useful for an OSS process; it allows obsolescent Guardian interfaces to interact with the process.

By default, this restriction is inherited by any child or successor process. The default can be overridden by using the **_TPC_IGNORE_FORCEPIN_ATTR** field.

**_TPC_IGNORE_FORCEPIN_ATTR**
Ignores the **_TPC_HIGHPIN_OFF** restriction specified for or inherited by the caller or parent process. When **_TPC_IGNORE_FORCEPIN_ATTR** is specified, the resulting process has a restricted PIN only if **_TPC_HIGHPIN_OFF** is also specified or if the object file for the program or a user library lacks the HIGHPIN attribute.

**_TPC_OVERRIDE_DEFMODE**
Specifies that the DEFINE mode of the new process is to be set according to the **_TPC_ENABLE_DEFINES** option rather than to the caller's current DEFINE mode.

**_TPC_PROCESS_DEFINES_ONLY**
Propagates only the current set of DEFINEs.

**_TPC_SUPPLIED_DEFINES_ONLY**
Propagates only the DEFINEs indicated by the **pe_defines** field.

**pe_debug_options**
Provides control over the selection between the default and symbolic debuggers and over the creation of the saveabend file. A saveabend file can be examined by using the symbolic debugger to determine the cause of the abnormal termination. In addition, you can use this option to force the new process to enter the default debugger before executing. Possible options are:

**_TPC_CODEFILE_INSPECT_SAVEABEND**
Uses the saveabend and INSPECT mode flags in the program file.

**_TPC_DEBUG_NOSAVE**
Uses the default debugger but does not create a saveabend file.

**_TPC_DEBUG_SAVEABEND**
Uses the default debugger and creates a saveabend file.

**_TPC_ENTER_DEBUG**
Starts the new process in the default debugging utility.

**_TPC_INSPECT_NOSAVE**
Uses the symbolic debugger but does not create a saveabend file.

**_TPC_INSPECT_SAVEABEND**
Uses the symbolic debugger and creates a saveabend file.

**pe_defines** Points to a specified saved set of DEFINEs created by using the Guardian DEFINESAVE procedure. These DEFINEs are propagated to the new process if either **_TPC_SUPPLIED_DEFINES_ONLY** or **_TPC_BOTH_DEFINES** is specified in the **pe_create_options** field.

**Note:** This string is not null-terminated.

**pe_defines_len**
Specifies the length of the string in the **pe_defines** field.

**pe_extswap_file_name**
Points to a null-terminated string specifying the name of a disk file in the Guardian file system to be used as the swap file for the extended data segment. For example, if the Guardian filename is $A.B.D, the name used is **/G/a/b/d**.

This file cannot have the same name as that of a file used in a preceding call to the **tdm_fork( )** function.

This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is checked for validity but otherwise ignored.

By default, the new process uses KMSF to manage its extended swap segment. HP recommends using the default.

**pe_heap_max** Specifies the maximum size of the heap in bytes for the new process if it is a native process.

See the *C/C++ Programmer's Guide* description of the **HEAP** pragma for guidance on the use of nonzero values for this field.

If a value is specified for this field for G-series TNS or accelerated object files, the specified value is ignored.

**pe_hometerm** Points to the null-terminated name in the Guardian file system for the home terminal. For example, if the Guardian name is $ztnt.#xyz, the name used is **/G/ztnt/#xyz**.

**pe_jobid** Specifies the job ID of the new process.

**pe_len** Specifies the size of the structure in bytes. This value is set by **#define DEFAULT_PROCESS_EXTENSION** and should not be changed.

**pe_library_name**
Points to the name of the user library to be bound to the new process. The string that is pointed to is null-terminated and in OSS name format. If the pointer points to a zero-length string (a NULL character), the new process runs with no user library. An equivalent call to the Guardian PROCESS_LAUNCH_ procedure does this by setting the library filename length to -1.

This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is ignored.

**pe_mainstack_max**
Specifies the maximum size of the main stack in bytes for the new process.

If the calling process specifies a value, the value must be less than 32 MB. If the calling process does not specify a value or specifies a 0 (zero) value, the value specified in the object file of the new process is used. If no value is specified in the object file, the default value of 1 MB (for TNS/R systems) or 2 MB (for TNS/E systems) is used.

**pe_memory_pages**
Specifies the size of the data stack in 2 KB units. This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is checked for validity but otherwise ignored.

**pe_name_options**
Specifies process naming as:

**_TPC_GENERATE_NAME**
The system generates the name.

**_TPC_NAME_SUPPLIED**
The process name is indicated by the **pe_process_name** field.

**_TPC_NO_NAME**
The new process is unnamed.

**pe_OSS_options**
Specifies OSS options. No special action on signals is the default and only current OSS option.

**pe_pfs_size** Specifies the size of the PFS for the new process (this field is ignored).

**pe_priority** Specifies the priority of the new process.

**pe_process_name**
Points to the null-terminated Guardian process name if **_TPC_NAME_SUPPLIED** is specified in the **pe_name_options** field. For example, if the Guardian process name is $DELM, the name used is **/G/delm**.

**pe_space_guarantee**
Specifies the minimum available swap space to guarantee for the new process.

If the calling process specifies a value, the value must be less than or equal to a multiple of the page size of the processor in which the new process will run. Values less than a multiple of the page size are rounded up to the next multiple of the page size. If the calling process does not specify a value or specifies a 0 (zero) value, the value specified in the native object file of the new process is used. If no value is specified in the native object file, the default value of 0 (zero) is used, and enough swap space is guaranteed to launch the process.

If the new process requires a guarantee of available swap space and the system cannot guarantee the required amount, the function call fails, and **errno** is set to the value of [EAGAIN].

If a value is specified for this field for G-series TNS or accelerated object files, the specified value is used for the main stack of the new process.

**pe_swap_file_name**
Points to a null-terminated string specifying the name of a file in the Guardian file system to be used as the swap file for the stack segment. For example, if the Guardian filename is $A.B.C, the name used is **/G/a/b/c**.

This file cannot have the same name as that of a file used in a preceding call to the **tdm_fork( )** function.

This field is not used in the current RVU of Open System Services. It exists for compatibility with older RVUs. Any specified value is checked for validity but otherwise ignored.

**pe_ver** Specifies the version of the **process_extension** structure. This value is set by **#define DEFAULT_PROCESS_EXTENSION** and should not be changed.

The MOM and ANCESTOR fields in the new process differ from those of a process created in the Guardian environment if the new process is named (the **pe_name_options** field is set to **_TPC_NAME_SUPPLIED** or **_TPC_GENERATE_NAME**). If the calling process is unnamed, the ANCESTOR field for the new process is set to the caller's MOM field, and the MOM field of the new process is null. If the calling process is named, the ANCESTOR field of the new process is set to the ANCESTOR field of the calling process, and the MOM field of the new process is null.

The MOM and ANCESTOR fields for the new process are the same as for a process created in the Guardian environment if the new process is unnamed (the **pe_name_options** field is set to **_TPC_NO_NAME**). If the caller is unnamed, the MOM field for the new process is set to the MOM field of the caller. If the caller is named, the MOM field for the new process is set to the ANCESTOR field of the calling process.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Output Structure Information

If the *pr_results* parameter does not contain a null pointer, it points to an output structure defined in the **tdmext.h** header file. This structure can contain fields that vary from RVU to RVU, including reserved and filler fields.

First, the output structure must be initialized by using the **#define DEFAULT_PROCESS_EXTENSION_RESULTS**. This initialization sets the value of the **pr_len** field to the correct value for the current RVU. The value of the **pr_len** field should not be modified after being set by **#define DEFAULT_PROCESS_EXTENSION_RESULTS**.

The **process_extension_results** output structure is described in the **process_extension_results(5)** reference page.

## RETURN VALUES

If the **tdm_execvep( )** function returns to the calling process image, an error has occurred; the return value is -1, and **errno** is set to indicate the error. If the *pr_results* parameter does not contain a null pointer, the structure it points to returns additional error information, including the PROCESS_LAUNCH_ error and error detail.

## ERRORS

If any of the following conditions occurs, the **tdm_execvep( )** function sets **errno** to the corresponding value, file descriptors marked close-on-exec are not closed, signals set to be caught are not set to the default action, and none of these are changed:

- The *argv*[ ] array of pointers
- The *envp*[ ] array of pointers
- The elements pointed to by these arrays
- The value of the global variable **environ**
- The pointers contained within the global variable **environ**
- The elements pointed to by **environ** pointers
- The effective user ID of the current process
- The effective group ID of the current process

[E2BIG] The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit. The limit can be obtained by calling the **sysconf(_SC_ARG_MAX)** function.

[EACCES] One of these conditions exists:

- Search permission is denied for the directory components of the pathname prefix to the process image file.
- The new process image file, any library file, or script file denies execution permission.
- The new process image file is not a regular file.

[EAGAIN] System resources such as disk space, process control block (PCB) space, MAP-POOL space, stack space, or PFS space are temporarily inadequate.

[EFAULT] An address for a parameter in the **process_extension** structure pointed to by *pe_parms* is out of allowable bounds. The Guardian PROCESS_LAUNCH_ error and error detail information is returned in the structure pointed to by the *pr_results* parameter, unless *pr_results* contains a null pointer.

[EHLDSEM] The process tried to create a new process in a different processor while having at least one **semadj** value.

[EINVAL] One of these conditions exists:

- An invalid parameter value was supplied in the **process_extension** structure pointed to by *pe_parms*. The Guardian PROCESS_LAUNCH_ error and error detail information is returned in the structure pointed to by the *pr_results* parameter, unless *pr_results* contains a null pointer.
- The new process image file is a binary executable file with invalid attributes.

[EIO] Some physical input or output error has occurred. Either a file cannot be opened because of an input or output error or data has been lost during an input or output transfer. This value is used only for errors on the object file of a loaded program or library, or during data transfer with a Guardian environment home terminal.

For systems running J06.07 and later J-series RVUs or H06.18 or later H-series RVUs, this error can also occur when the OSS file system is out of memory and one or more open files cannot be propagated from the parent process to the child process. In this case, if you are running a program from the shell with the shell reporting any errors, you might see an error like this:

/bin/-sh: /bin/ps: tdm_execve(): failed with unexpected error pr_errno=(4005) pr_TPCerror=(110) pr_TPCdetail=(36)

where:

- **pr_errno** is the [EIO] error
- **pr_TPCerror** is the Guardian PROCESS_LAUNCH_ or PROCESS_CREATE_ error.

[ELOOP] Too many symbolic links were encountered in pathname resolution.

[EMFILE] The maximum number of files are open. The process attempted to open more than the maximum number of file descriptors allowed for the process. The process file segment (PFS) of the new process might be smaller than that of the calling process.

[ENAMETOOLONG]
One of these is too long:

- The pathname pointed to by the *file* parameter
- A component of the pathname pointed to by the *file* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the value specified by the *file* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOCPU] The selected processor does not exist, or the selected processor is down or otherwise unavailable for process creation.

[ENODEV] The system cannot find the file system containing the process image file.

[ENOENT] One of these conditions exists:

- One or more components of the new process image file's pathname do not exist.
- The *file* parameter points to an empty string.

[ENOEXEC] The command interpreter could not be invoked following failure to execute the process image file identified by the *file* parameter.

[ENOMEM] Required resources are not available. Subsequent calls to the same function will not succeed for the same reason.

Possible causes of this error include insufficient primary memory (stack, globals, or heap) for the new process.

[ENOTDIR] A component of the path prefix of the new process image file is not a directory.

[ENOTOSS] The calling process is not an OSS process. The **tdm_execvep( )** function cannot be used from the Guardian environment.

[EPERM] One of the following conditions exist:

- The calling process does not have appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[ETXTBSY] The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.

[EUNKNOWN]
Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

**RELATED INFORMATION**

Commands: **eld(1)**, **ld(1)**, **nld(1)**.

Functions: **alarm(3)**, **chmod(2)**, **exec(2)**, **_exit(2)**, **exit(3)**, **fcntl(2)**, **fork(2)**, **getenv(3)**, **putenv(3)**, **semget(2)**, **shmat(2)**, **sigaction(2)**, **system(3)**, **tdm_execve(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **times(3)**, **ulimit(2)**, **umask(2)**.

Files: **signal(4)**.

Miscellaneous: **environ(5)**, **process_extension_results(5)**.

**STANDARDS CONFORMANCE**

This function is an extension to the XPG4 Version 2 specification.

**NAME**

**tdm_fork -** Creates a new process with HP extensions

**LIBRARY**

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

**SYNOPSIS**

**#include <tdmext.h>**

**pid_t tdm_fork(**
**const struct process_extension** **pe_parms***,**
**struct process_extension_results** **pr_results***);**

**PARAMETERS**

*pe_parms*
Points to the input structure containing Guardian process attributes to be assigned to the child process. The structure is defined in the **tdmext.h** header file.

When this parameter contains a null pointer, the **tdm_fork( )** function assumes default Guardian attributes. Otherwise, the structure must be defined locally and initialized before its first use. Initialization is done using the **#define DEFAULT_PROCESS_EXTENSION**, as defined in the **tdmext.h** header file. The initialized values can then be modified as appropriate for the call.

*pr_results*
Points to the output structure containing optional process identification and error information. On successful return, this information includes the Guardian process handle and OSS process ID (PID) of the process. If the call is not successful, the OSS error number and Guardian PROCESS_LAUNCH_ procedure error and error detail are returned in this structure. The structure is defined in the **tdmext.h** header file.

The structure must be defined locally and initialized before its first use. Initialization is done using the **#define DEFAULT_PROCESS_EXTENSION_RESULTS**, as defined in the **tdmext.h** header file.

See the **process_extension_results(5)** reference page for information about the content of the structure. The **tdmext.h** header file is not kept current when new error codes are defined for process creation functions. The list of **_TPC_** macros described in that reference page is not complete; for a current description of error macros and error codes, see the Guardian header file $SYSTEM.ZSPIDEF.ZGRDC or the summary of process-creation errors in the *Guardian Procedure Calls Reference Manual* (see the table entitled "Summary of Process Creation Errors").

**DESCRIPTION**

The **tdm_fork( )** function creates a child OSS process. The created process is referred to as the child and the caller as the parent. The child process executes the same program file as the parent. The child process retains many of its parent's OSS process attributes and obtains system-derived values for others. For Guardian process attributes, the child process can retain default values or can have values specified in the **tdm_fork( )** call.

### Use From the Guardian Environment

If called from a Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

### OSS Attributes

The child process inherits the following OSS attributes from the parent process:

- Environment
- Close-on-exec flags
- Signal-handling settings
- Saved-set-user-ID mode bit
- Saved-set-group-ID mode bit
- Process group ID
- Current directory
- Root directory
- File mode creation mask
- File size limit (see the **ulimit(2)** reference page)
- Attached semaphore set IDs
- Attached shared memory segments

The OSS attributes of the child process differ from those of the parent process in the following ways:

- The child process has a unique OSS process ID (PID) and does not match any active process group ID.
- The parent process ID of the child process matches the OSS process ID of the parent.
- The child process has its own copy of the parent process's file descriptors. However, each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent process.
- The child process does not inherit any file open created by a Guardian function or procedure call.
- The child process does not inherit file locks.
- The child process's **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** values are set to 0 (zero).
- Any pending alarms are cleared in the child process.
- Any signals pending for the parent process are not inherited by the child process.
- Any adjust-on-exit (**semadj**) values of the parent process are not inherited by the child process.

- The child process shares directory streams with the parent. They share the same block of directory entries. When reading an entry, the buffer pointer is advanced by one entry. From the perspective of either process, an entry might be skipped.

  If both processes call the **readdir( )** function for a shared stream, the results are undefined. After such a call by both functions, another call to the **readdir( )** function by either process has undefined results.

**Default Guardian Attributes**

If the *pe_parms* parameter contains a null pointer, then the child process inherits all the following default Guardian attributes from the parent process:

- Program file
- Any library file
- The size and contents of any instance data segments for native libraries
- Priority (the child process inherits the parent's current priority)
- Processor on which the process executes
- Home terminal
- For G-series TNS processes and accelerated processes, the size and contents of the data segment
- For G-series TNS processes and accelerated processes, the size and contents of the extended data segment

  The assignment of the data segment size is different from the assignment made when creating a child process with Guardian procedures.
- For native processes, the contents of the stack segment from its origin to the currently in-use location; the rest of the child process stack is 0 (zero)
- For native processes, the size and contents of the globals-heap segment
- Job ID
- DEFINE mode
- Creator access ID (CAID)
- Process access ID (PAID)
- Security group list
- Job ancestor or GMOM
- Unread system message index (PCBMCNT)

  This attribute assignment is different from the assignment made when creating a child process with Guardian procedures.
- Outstanding incoming and outgoing message limits

  This attribute assignment is different from the assignment made when creating a child process with Guardian procedures.

- Login, remote login, and saveabend flags
- File creation mask
- System debugger selection (based on Inspect mode and OSS read access rights on the program file)

If the *pe_parms* parameter contains a null pointer, then the default Guardian attributes of the child process differ from those of the parent process in the following ways:

- Segments created or shared using Guardian procedures such as SEGMENT_ALLOCATE_ are not inherited.
- The child process does not inherit the parent process extended swap file (if any). For a G-series TNS process or an accelerated process, the extended data segment is managed by the Kernel Managed Storage Facility (KMSF) unless an extended swap file is specified in the **pe_extswap_file_name** field of the **process_extension** structure described elsewhere in this reference page.
- The child's process name is system-generated if the RUNNAMED option is set in the program file. Otherwise, the process is unnamed.
- The DEFINEs inheritance for the child is based on the parent's DEFINE mode.
- The process identification number (PIN) of the child process is unrelated to that of the parent process. Usually, the PIN of the child process is unrestricted. However, the PIN can be restricted to the range 0 through 254 under the following conditions:
  - — The HIGHPIN flag is not set in, or is absent from, the program file or any library file.
  - — **_TPC_HIGHPIN_OFF** is specified in the **pe_create_options** field of the **process_extension** structure, described elsewhere in this reference page.
  - — The restriction is inherited. See **_TPC_IGNORE_FORCEPIN_ATTR** in the **pe_create_options** field of the **process_extension** structure, described elsewhere in this reference page, for more information about controlling inheritance.
- The MOM field for the child process is set to 0 (zero).
- Code breakpoints and memory breakpoints are not inherited.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Setting Guardian Attributes

The input structure pointed to by the *pe_parms* parameter permits the setting of Guardian attributes for the child process.

First, the input structure must be initialized to the default values (see **Default Guardian Attributes**, earlier in this reference page) using the **#define DEFAULT_PROCESS_EXTENSION**. After the data structure is initialized, the values can be set using literals that are defined in the **tdmext.h** header file.

If any optional parameter specified in the structure pointed to by *pe_parms* is not passed, the child process assumes the corresponding default value.

The input structure is defined in the **tdmext.h** header file. This structure can contain fields that vary from release to release, including reserved and filler fields.

The following fields are meaningful:

```
#if defined (__LP64) || defined (_PROCEX32_64BIT)
   typedef struct process_extension {
           short   pe_ver;
           short   pe_len;
           int     pe_pfs_size;
           long long   pe_mainstack_max;
           long long   pe_heap_max;
           long long   pe_space_guarantee;
           char _ptr64  *pe_library_name;
           char _ptr64  *pe_swap_file_name;
           char _ptr64  *pe_extswap_file_name;
           char _ptr64  *pe_process_name;
           char _ptr64  *pe_hometerm;
           char _ptr64  *pe_defines;
           short   pe_defines_len;
           short   pe_priority;
           short   pe_cpu;
           short   pe_memory_pages;
           short   pe_jobid;
           short   pe_name_options;
           short   pe_create_options;
           short   pe_debug_options;
           short   pe_OSS_options;
           char    filler_1[6];
   } process_extension_def;
#else  /* !defined (__LP64) && !defined (_PROCEX32_64BIT) */
   typedef struct process_extension {
           long    pe_len;
           char    *pe_library_name;
           char    *pe_swap_file_name;
           char    *pe_extswap_file_name;
           short   pe_priority;
           short   pe_cpu;
           short   pe_name_options;
           char    filler_1[2];
           char    *pe_process_name;
           char    *pe_hometerm;
           short   pe_memory_pages;
           short   pe_jobid;
           short   pe_create_options;
           char    filler_2[2];
           char    *pe_defines;
           short   pe_defines_len;
           short   pe_debug_options;
           long    pe_pfs_size;
           short   pe_OSS_options;
           char    filler_3[2];
           long    pe_mainstack_max;
           long    pe_heap_max;
           long    pe_space_guarantee;
   } process_extension_def;
```

**#endif /* !defined (__LP64) && !defined (_PROCEX32_64BIT) */**

When an application is compiled in 64-bit compile mode or compiled using the **#define _PROCEX32_64BIT 1** feature test macro or an equivalent compiler command option, the application will use the version of the **process_extension** structure that contains 64-bit data types. The **_PROCEX32_64BIT** flag is only required if a 32-bit process must specify larger 64-bit values for **pe_mainstack_max**, **pe_heap_max**, and **pe_space_guaranter**. These larger data types are optional when creating a 64-bit process.

**Note:** The input structure supports two versions: one that contains 64-bit data types and one that contains 32-bit data types. Because the order in which the fields appear in this structure varies significantly based on the version in use, the field definitions below are defined alphabetically instead of sequentially.

The input structure passes this information:

**pe_cpu**
Specifies the processor on which the new process will execute. However, -1 (default value) is the only valid value accepted, which means create the child process on the same CPU as that of the parent.

**pe_create_options**
Specifies process creation options as:

**_TPC_BOTH_DEFINES**
Propagates the current DEFINEs and the DEFINEs indicated in the input structure.

**_TPC_ENABLE_DEFINES**
Enables DEFINEs when set if **_TPC_OVERRIDE_DEFMODE** is also set. Disables DEFINEs when not set.

**_TPC_HIGHPIN_OFF**
Restricts the new process to a PIN in the range 0 through 254. This restriction is rarely useful for an OSS process; it allows obsolescent Guardian interfaces to interact with the process.

By default, this restriction is inherited by any child or successor process. The default can be overridden by using the **_TPC_IGNORE_FORCEPIN_ATTR** field.

**_TPC_IGNORE_FORCEPIN_ATTR**
Ignores the **_TPC_HIGHPIN_OFF** restriction specified for or inherited by the caller or parent process. When **_TPC_IGNORE_FORCEPIN_ATTR** is specified, the resulting process has a restricted PIN only if **_TPC_HIGHPIN_OFF** is also specified or if the object file for the program or a user library lacks the HIGHPIN attribute.

**_TPC_OVERRIDE_DEFMODE**
Specifies that the DEFINE mode of the new process is to be set according to the **_TPC_ENABLE_DEFINES** option rather than to the caller's current DEFINE mode.

**_TPC_PROCESS_DEFINES_ONLY**
Propagates only the current set of DEFINEs.

**_TPC_SUPPLIED_DEFINES_ONLY**
Propagates only the DEFINEs indicated by the **pe_defines** field.

**pe_debug_options**
Provides control over the selection between the default and symbolic debuggers and over the creation of the saveabend file. A saveabend file can be examined by using the symbolic debugger to determine the cause of the abnormal termination. In addition, you can use this option to force the new process to enter the default debugger before executing. Possible options are:

**_TPC_CODEFILE_INSPECT_SAVEABEND**
Uses the saveabend and INSPECT mode flags in the program file.

**_TPC_DEBUG_NOSAVE**
Uses the default debugger but does not create a saveabend file.

**_TPC_DEBUG_SAVEABEND**
Uses the default debugger and creates a saveabend file.

**_TPC_ENTER_DEBUG**
Starts the new process in the default debugging utility.

**_TPC_INSPECT_NOSAVE**
Uses the symbolic debugger but does not create a saveabend file.

**_TPC_INSPECT_SAVEABEND**
Uses the symbolic debugger and creates a saveabend file.

**pe_defines** Points to a specified saved set of DEFINEs created by using the Guardian DEFINESAVE procedure. These DEFINEs are propagated to the new process if either **_TPC_SUPPLIED_DEFINES_ONLY** or **_TPC_BOTH_DEFINES** is specified in the **pe_create_options** field.

**Note:** This string is not null-terminated.

**pe_defines_len**
Specifies the length of the string in the **pe_defines** field.

**pe_extswap_file_name**
Points to a null-terminated string specifying the name of a disk file in the Guardian file system to be used as the swap file for the extended data segment. For example, if the Guardian filename is $A.B.D, the name used is **/G/a/b/d**.

This file cannot have the same name as that of a file used in a preceding call to the **tdm_fork( )** function.

This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is checked for validity but otherwise ignored.

By default, the new process uses KMSF to manage its extended swap segment. HP recommends using the default.

**pe_heap_max** Specifies the maximum size of the heap in bytes for the new process if it is a native process.

See the *C/C++ Programmer's Guide* description of the **HEAP** pragma for guidance on the use of nonzero values for this field.

If a value is specified for this field for G-series TNS or accelerated object files, the specified value is ignored.

**pe_hometerm** Points to the null-terminated name in the Guardian file system for the home terminal. For example, if the Guardian name is $ztnt.#xyz, the name used is **/G/ztnt/#xyz**.

**pe_jobid** Specifies the job ID of the new process.

**pe_len** Specifies the size of the structure in bytes. This value is set by **#define DEFAULT_PROCESS_EXTENSION** and should not be changed.

**pe_library_name**
Points to the name of the user library to be bound to the new process. The string that is pointed to is null-terminated and in OSS name format. If the pointer points to a zero-length string (a NULL character), the new process runs with no user library. An equivalent call to the Guardian PROCESS_LAUNCH_ procedure does this by setting the library filename length to -1.

This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is ignored.

**pe_mainstack_max**
Specifies the maximum size of the main stack in bytes for the new process.

If the calling process specifies a value, the value must be less than 32 MB. If the calling process does not specify a value or specifies a 0 (zero) value, the value specified in the object file of the new process is used. If no value is specified in the object file, the default value of 1 MB (for TNS/R systems) or 2 MB (for TNS/E systems) is used.

**pe_memory_pages**
Specifies the size of the data stack in 2 KB units. This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is checked for validity but otherwise ignored.

**pe_name_options**
Specifies process naming as:

**_TPC_GENERATE_NAME**
The system generates the name.

**_TPC_NAME_SUPPLIED**
The process name is indicated by the **pe_process_name** field.

**_TPC_NO_NAME**
The new process is unnamed.

**pe_OSS_options**
Specifies OSS options. No special action on signals is the default and only current OSS option.

**pe_pfs_size** Specifies the size of the PFS for the new process (this field is ignored).

**pe_priority** Specifies the priority of the new process.

**pe_process_name**
Points to the null-terminated Guardian process name if **_TPC_NAME_SUPPLIED** is specified in the **pe_name_options** field. For example, if the Guardian process name is $DELM, the name used is **/G/delm**.

**pe_space_guarantee**
Specifies the minimum available swap space to guarantee for the new process.

If the calling process specifies a value, the value must be less than or equal to a multiple of the page size of the processor in which the new process will run. Values less than a multiple of the page size are rounded up to the next multiple of the page size. If the calling process does not specify a value or specifies a 0 (zero) value, the value specified in the native object file of the new process is used. If no value is specified in the native object file, the default value of 0 (zero) is used, and enough swap space is guaranteed to launch the process.

If the new process requires a guarantee of available swap space and the system cannot guarantee the required amount, the function call fails, and **errno** is set to the value of [EAGAIN].

If a value is specified for this field for G-series TNS or accelerated object files, the specified value is used for the main stack of the new process.

**pe_swap_file_name**
Points to a null-terminated string specifying the name of a file in the Guardian file system to be used as the swap file for the stack segment. For example, if the Guardian filename is $A.B.C, the name used is **/G/a/b/c**.

This file cannot have the same name as that of a file used in a preceding call to the **tdm_fork( )** function.

This field is not used in the current RVU of Open System Services. It exists for compatibility with older RVUs. Any specified value is checked for validity but otherwise ignored.

**pe_ver** Specifies the version of the **process_extension** structure. This value is set by **#define DEFAULT_PROCESS_EXTENSION** and should not be changed.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Output Structure Information

If the *pr_results* parameter does not contain a null pointer, it points to an output structure defined in the **tdmext.h** header file. This structure can contain fields that vary from release to release, including reserved and filler fields.

First, the output structure must be initialized using the **#define DEFAULT_PROCESS_EXTENSION_RESULTS**. This initialization sets the value of the **pr_len** field to the correct value for the current release. The value of the **pr_len** field should not be modified after being set by **#define DEFAULT_PROCESS_EXTENSION_RESULTS**.

The **process_extension_results** output structure is described in the **process_extension_results(5)** reference page.

### Shared Memory

Any attached shared memory segments are attached to both the child process and the parent process when both processes execute in the same processor. Any attached shared memory segments are detached from the child process by a successful call to the **tdm_fork( )** function when the child process executes in a different processor than that used by the parent. Refer to the **shmat(2)** reference page for additional information about shared memory segment use.

### Semaphores

Semaphore set IDs attached to a parent process are also attached to the child process if the child process executes in the same processor as the parent.

A semaphore set cannot be shared when a **semadj** value exists for the parent process and the child process is created in a different processor. When that condition exists, a call to the **tdm_fork( )** function fails and **errno** is set to [EHLDSEM].

Refer to the **semget(2)** reference page for additional information about semaphore use.

### Open Files

File descriptors open in the parent process remain open in the child process, except for those opened using a Guardian function or procedure call. For those file descriptors that remain open, all attributes of the open file descriptor, including file locks, remain unchanged.

### Open Pipes and FIFOs

A pipe or FIFO associated with an open file descriptor in the parent process remains connected in the child process. If the child process runs in a different processor than the parent process, the processor that runs the child process must also be running an OSS pipe server process.

If no OSS pipe server process is running in the new processor, the child process cannot use the pipe or FIFO; calls specifying the file descriptor for the pipe or FIFO fail with **errno** set to [EWRONGID]. The child process can only close the invalid file descriptor.

### Existing Sockets

A socket associated with an open file descriptor in the parent process remains connected in the child process. If the child process runs in a different processor than the parent process, the processor that runs the child process must also be running a socket transport agent process.

If no socket transport agent process is running in the new processor, the child process cannot use the socket; calls specifying the file descriptor for the socket fail with **errno** set to [EWRONGID]. The child process can only close the invalid file descriptor.

### Sharing Guardian Files

After a successful call to the **tdm_fork( )** function, the initial position within an open EDIT file (file code 101) in the Guardian file system (a file in **/G**) that was opened by a call to the OSS **open( )** function is the same for both the parent and child processes. However, the position is not shared; that is, changing the position used by one process does not change the position used by the other process.

### Floating-Point Data

If the parent process uses IEEE floating-point data, the child process inherits all the floating-point register contents of the parent process and any computation that was started before the **tdm_fork( )** function call finishes in the child process. The contents of the status and control register also are inherited.

## RETURN VALUES

Upon successful completion, the **tdm_fork( )** function returns the value 0 (zero) to the child process and returns the OSS process ID of the child process to the parent process. If the *pr_results* parameter does not contain a null pointer, it returns the Guardian process handle of the child process in addition to the OSS process ID.

If the **tdm_fork( )** function fails, the value -1 is returned to the parent process, no child process is created, and **errno** is set to indicate the error. If *pr_results* does not contain a null pointer, it returns additional error information including the PROCESS_LAUNCH_ procedure error and error detail.

## ERRORS

If any of the following conditions occurs, the **tdm_fork( )** function sets **errno** to the corresponding value:

[EACCES] Open for execute access on the code file or any library file was denied.

[EAGAIN] System resources such as disk space, process control block (PCB) space, MAPPOOL space, stack space, or PFS space are temporarily inadequate.

[EFAULT] An address for a parameter in the **process_extension** structture pointed to by *pe_parms* is out of allowable bounds. The Guardian PROCESS_LAUNCH_ procedure error and error detail information is returned in the structure pointed to by the *pr_results* parameter, unless *pr_results* contains a null pointer.

[EHLDSEM] The process tried to create a child process in a different processor while having at least one **semadj** value.

[EINVAL] An invalid parameter value was supplied in the **process_extension** structure pointed to by *pe_parms*. The Guardian PROCESS_LAUNCH_ error and error detail information is returned in the structure pointed to by the *pr_results* parameter, unless *pr_results* contains a null pointer.

[EIO] Some physical input or output error has occurred. Either a file cannot be opened because of an input or output error or data has been lost during an input or output transfer. This value is used only for errors on the object file of a loaded program or library, or during data transfer with a Guardian environment home terminal.

[ENOMEM] Required resources are not available. Subsequent calls to the same function will not succeed for the same reason.

Possible causes of this error include insufficient primary memory (stack, globals, or heap) for the child process.

[ENOTOSS] The parent process is not an OSS process. The **tdm_fork( )** function cannot be used from the Guardian environment.

[EUNKNOWN] Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

## RELATED INFORMATION

Functions: **exec(2)**, **_exit(2)**, **fork(2)**, **raise(3)**, **semget(2)**, **semop(2)**, **shmat(2)**, **sigaction(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **times(3)**, **ulimit(2)**, **umask(2)**, **wait(2)**.

Miscellaneous: **process_extension_results(5)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification.

## NAME

**tdm_spawn -** Executes a new process with HP extensions

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yosskdll**

## SYNOPSIS

```
#include &lt;spawn.h&gt;
#include &lt;tdmext.h&gt;

[ extern char **environ; ]

pid_t tdm_spawn(
        const char *path,
        const int fd_count,
        const int fd_map[ ],
        const struct inheritance *inherit,
        char * const argv[ ],
        char * const envp[ ],
        const struct process_extension *pe_parms,
        struct process_extension_results *pr_results);
```

## PARAMETERS

****environ** Points to an array of character pointers to environment strings. The environment strings define the OSS environment for the parent process. The **environ** array is terminated by a null pointer.

*path* Points to a null-terminated string containing a pathname that identifies the new process image file. The pathname is absolute if it starts with a slash (/) character. Otherwise, the pathname is relative and is resolved by prefixing the current working directory.

*fd_count* Specifies the number of file descriptors designated by the *fd_map*[ ] parameter. All file descriptors higher than *fd_count* are closed in the new process. This parameter can take values from 0 (zero) through **POSIX_OPEN_MAX**.

*fd_map*[ ] Maps file descriptors from the parent process to the new process. File descriptors identified with the value **SPAWN_FDCLOSED** are closed in the new process.

If this parameter is a null pointer, all open OSS file descriptors of the parent process (except for files opened by Guardian function or procedure calls and those with the **FD_CLOEXEC** attribute flag set) are inherited by the new process. Such inherited file descriptors behave here as they do for the **tdm_execve( )** function.

*inherit* Points to a structure that allows the process group ID and signal mask of the new process to be specified in addition to a list of signals that the new process will take default action on. The structure is defined in the **spawn.h** header file.

*argv*[ ] Specifies an array of character pointers to null-terminated strings containing arguments to be passed to the main function of the new program. *argv***[0]** should point to the null-terminated string containing the filename of the new process image. The last member of this array must be a null pointer.

*envp***[ ]** Specifies an array of character pointers to null-terminated strings that describe the environment for the new process.

*pe_parms* Points to the input structure containing Guardian process attributes to be assigned to the new process. The structure is defined in the **tdmext.h** header file.

When this parameter contains a null pointer, the **tdm_spawn( )** function assumes default Guardian attributes. Otherwise, the structure must be defined locally and initialized before its first use. Initialization is done using the **#define DEFAULT_PROCESS_EXTENSION**, as defined in the **tdmext.h** header file. The initialized values can then be modified as appropriate for the call.

*pr_results* Points to the output structure containing optional process identification and error information. In case of error, this structure provides additional information, including the PROCESS_LAUNCH_ procedure error and error detail. The structure is defined in the **tdmext.h** header file.

The structure must be defined locally and initialized before its first use. Initialization is done by using the **#define DEFAULT_PROCESS_EXTENSION_RESULTS**, as defined in the **tdmext.h** header file.

See the **process_extension_results(5)** reference page for information about the content of the structure. The **tdmext.h** header file is not kept current when new error codes are defined for process creation functions. The list of **_TPC_** macros described in that reference page is not complete; for a current description of error macros and error codes, see the Guardian header file $SYSTEM.ZSPIDEF.ZGRDC or the summary of process-creation errors in the *Guardian Procedure Calls Reference Manual* (see the table entitled "Summary of Process Creation Errors").

## DESCRIPTION

The **tdm_spawn( )** function creates a new process image. The new image is constructed from a regular executable file, called a new process image file. The new process image file is formatted as an executable text or binary file in one of the formats recognized by the **tdm_spawn( )** function.

The **tdm_spawn( )** function is similar to the **tdm_spawnp( )** function. The main difference is the way the pathname for the process image file is resolved. **tdm_spawn( )** always resolves relative pathnames by prefixing the current working directory; see **Identifying the Process Image File**, later in this reference page. **tdm_spawnp( )** sometimes uses the **PATH** environment variable to resolve pathnames.

The **tdm_spawn( )** function provides a different way to create a new process than the way provided by the **tdm_fork( )** and **tdm_execve( )** functions. **tdm_spawn( )** provides a more efficient way to create a new process to execute a new program file. However, **tdm_spawn( )** does not provide all the function provided by **tdm_fork( )** and **tdm_execve( )**.

When a program is executed as a result of a **tdm_spawn( )** call, it is entered as a function call:

```
int main(
        int argc,
        char *argv[ ],
        char *envp);
```

Here, the *argc* parameter is the argument count, the *argv***[ ]** parameter is an array of character pointers to the arguments themselves, and the *envp* parameter is a pointer to a character array listing the environment variables. The *argv***[ ]** array is terminated by a null pointer. The null pointer is not counted in *argc*.

The arguments specified by a program using the **tdm_spawn( )** function are passed on to the new process image in the corresponding arguments to the **main( )** function.

**Use From the Guardian Environment**

If called from a Guardian process, the actions of this function are undefined, and **errno** is set to [ENOTOSS].

**Identifying the Process Image File**

The **tdm_spawn( )** function uses the *path* parameter to identify the process image file. This parameter points to the absolute pathname if the pathname starts with a slash ( / ) character. Otherwise, the pathname is relative and is resolved by prefixing the current working directory.

**Passing the Arguments**

The *argv***[ ]** parameter is an array of character pointers to null-terminated strings. The last member of this array is a null pointer. These strings constitute the argument list available to the new process image. The value in *argv***[0]** should point to a filename that is associated with the process being started by the **tdm_spawn( )** function.

**Specifying the Environment**

The *envp***[ ]** parameter is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated with a null pointer.

The number of bytes available for the new process's combined argument and environment lists has a system-imposed limit. This limit, which includes the pointers and the null terminators on the strings, is available by calling the **sysconf(_SC_ARG_MAX)** function.

**Executing a Binary File**

If the file specified as the new process image file is a binary executable file, the **tdm_spawn( )** function loads the file directly.

**Executing a Text File**

If the file specified as the new process image file is not a binary executable file, the **tdm_spawn( )** function examines the file to determine whether it is an executable text file. It checks for a header line in this format:

**#!** *interpreter_name* [*optional_string*]

The **#!** notation identifies the file as an executable text file. The new process image filename is constructed from the process image filename in the *interpreter_name* string, treating it like the *path* parameter. The Guardian input and output structures pointed to by the *pe_parms* and *pr_results* parameters apply to the command interpreter as they would to any process file.

The arguments passed to the new process are modified as listed:

- *argv***[0]** is set to the name of the command interpreter.
- If the *optional_string* portion is present, *argv***[1]** is set to *optional_string*.
- The next element of *argv***[ ]** is set to the original value of *path*.
- The remaining elements of *argv***[ ]** are set to the original elements of *argv***[ ]**, starting with *argv***[1]**. The original *argv***[0]** is discarded.

The **S_ISUID** and **S_ISGID** mode bits of an executable text file are ignored.

### When the File Is Invalid

If the process image file is not a valid executable object, or if the text file does not contain the header line, the **tdm_spawn( )** function returns and sets **errno** to [ENOEXEC].

### Open Files

The *fd_count* and *fd_map***[ ]** parameters determine which file descriptors that were open in the calling process remain open in the new process.

*fd_count* specifies the number of file descriptors to be designated by the *fd_map***[ ]** parameter.

*fd_map***[ ]** specifies how file descriptors in the parent process map to file descriptors in the new process. That is, the file descriptor in *fd_map***[0]** is copied to file descriptor 0 (zero) in the new process, the file descriptor in *fd_map***[1]** is copied to file descriptor 1 in the new process, and so on. If *fd_map***[ ]** has a null value, the *fd_count* parameter is ignored and all open file descriptors in the parent (except for files opened by Guardian function or procedure calls and those with the **FD_CLOEXEC** attribute flag set) are inherited without mapping by the new process. Such inherited file descriptors behave here as they do for the **tdm_execve( )** function.

If *fd_map***[ ]** does not have a null value, file descriptors from *fd_count* to **OPEN_MAX** are closed in the new process, as are entries in *fd_map***[ ]** that are identified with the value **SPAWN_FDCLOSED**.

If a file descriptor specified in *fd_map***[ ]** is invalid, the function call fails. (Any file descriptor created by a Guardian function or procedure call is invalid.) The **errno** variable is set to [EBADF].

For a G-series TNS process image or an accelerated process image only, if the process file segment (PFS) of the new process image is smaller than the process file segment of the calling process image and if the calling process image has a large number of file descriptors open, then the system might not be able to propagate all the open file descriptors to the new process image. When this situation occurs, the function call fails, and **errno** is set to the value of [EMFILE].

### Open Pipes and FIFOs

A pipe or FIFO associated with an open file descriptor in the parent process remains connected in the child process. If the child process runs in a different processor than the parent process, the processor that runs the child process must also be running an OSS pipe server process.

If no OSS pipe server process is running in the new processor, the child process cannot use the pipe or FIFO; calls specifying the file descriptor for the pipe or FIFO fail with **errno** set to [EWRONGID]. The child process can only close the invalid file descriptor.

### Existing Sockets

A socket associated with an open file descriptor in the calling process remains connected in the new process when the new process runs in the same processor as the calling process.

When the new process runs in a different processor than the calling process, the processor that runs the new process must also be running a socket transport agent process. If no socket transport agent process is running in the new processor, the new process cannot use the socket; calls specifying the file descriptor for the socket fail with **errno** set to [EWRONGID]. The new process can only close the invalid file descriptor.

### Sharing Guardian Files

After a successful call to the **tdm_spawn( )** function, the initial position within an open EDIT file (file code 101) in the Guardian file system (a file in **/G**) that was opened by a call to the OSS **open( )** function is the same for both the parent and child processes. However, the position is not shared; that is, changing the position used by one process does not change the position used by the other process.

**Shared Memory**

Any attached shared memory segments are detached from the child process by a successful call to the **tdm_spawn( )** function. See the **shmat(2)** reference page for additional information about shared memory segment use.

**Semaphores**

Semaphore set IDs attached to a parent process are also attached to the child process if the child process executes in the same processor as the parent.

A semaphore set cannot be shared when a **semadj** value exists for the parent process and the child process is created in a different processor. When that condition exists, a call to the **tdm_spawn( )** function fails and **errno** is set to [EHLDSEM].

See the **semget(2)** reference page for additional information about semaphore use.

**Signals**

The setting of signaling attributes in the new process depends on the information provided in the **inheritance** structure (pointed to by the *inherit* parameter).

This default signal information applies to the child process unless modified by the information in the **inheritance** structure:

- Signals set to the default action (**SIG_DFL**) in the parent process are set to the default action in the child process.
- Signals set to be ignored (**SIG_IGN**) by the parent process are set to be ignored by the child process.
- Signals that cause abnormal termination (**SIG_ABORT**) in the calling process image are set to that action in the new process image.
- Signals that cause entry into the debugger (**SIG_DEBUG**) in the calling process image are set to that action in the new process image.
- Signals set to be caught by the parent process are set to the default action in the child process (see the **signal(4)** reference page).
- The signal mask in the child process is inherited from the parent process.
- Signals pending in the parent process are disregarded by the child process.

The **inheritance** structure can modify the default signal information as listed:

- If the **SPAWN_SETSIGMASK** bit is set in *inherit*->**flags**, then *inherit*->**sigmask** contains the signal mask for the child process.
- If the **SPAWN_SETSIGDEF** bit is set in *inherit*->**flags**, then *inherit*->**sigdefault** specifies the signal set that is forced to the default action in the child process. Additional signals that are set to the default action in the parent process, or for which the parent process has a signal-catching function installed, are also set to the default action in the child process.

**Process Group**

By default, the child process is a member of the same process group as the parent. However, the new process can be designated a member of some other process group by setting the **SPAWN_SETPGROUP** bit in *inherit*->**flags**. The *inherit*->**pgroup** field specifies the process group number, or it contains the **SPAWN_NEWPGROUP** symbolic constant if the new process is to be the leader of a new process group.

**User ID and Group ID**

If the set-user-ID mode bit (**S_ISUID**) of the new process image file is set (see the **chmod(2)** reference page), the effective user ID of the new process image is set to the user ID of the owner of the new process image file. Similarly, if the set-group-ID mode bit (**S_ISGID**) of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved-set user ID and the saved-set group ID) for use by the **setuid( )** function.

**OSS Attributes**

These OSS attributes of the calling process image are unchanged after successful completion of the **tdm_spawn( )** function:

- Real user ID
- Real group ID
- Session membership
- Current working directory
- Root directory
- File mode creation mask (see the **umask(2)** reference page)
- File size limit (see the **ulimit(2)** reference page)

The OSS attributes of the child process differ from those of the parent process in these ways:

- The child process has a unique OSS process ID (PID) and does not match any active process group ID.
- The parent process ID of the child process matches the OSS process ID of the parent.
- The child process has its own copy of a subset of the parent process's file descriptors. See **Open Files**, earlier in this reference page. However, each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent process.
- The child process does not inherit file opens created by Guardian function or procedure calls.
- The child process does not inherit file locks.
- The child process's **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** values are set to 0 (zero).
- Any pending alarms are cleared in the child process.
- Any adjust-on-exit (**semadj**) values of the parent process are not inherited by the child process.
- Any signals pending for the parent process are not inherited by the child process.
- The signal mask of the child process is that of the parent process unless modified by the *inherit*->**sigmask** field. See **Signals**, earlier in this reference page.

- The set of signals for which default action is set and the set of signals to be ignored are the same in the child process as in the parent process unless modified by *inherit*->**sigdefault**. See **Signals**, earlier.
- The child process does not share directory streams with the parent. All open directory streams are closed for the child process.

**Default Guardian Attributes**

If the *pe_parms* parameter contains a null pointer, the newly created OSS process retains all of these default Guardian attributes of the process that calls the **tdm_spawn( )** function:

- Priority
- Processor on which the process executes
- Home terminal
- Job ID
- DEFINE mode switch
- Creator access ID (CAID)
- Process access ID (PAID), unless the **S_ISUID** mode bit of the new process image file is set
- Security group list
- Job ancestor or GMOM
- Unread system message index (PCBMCNT)

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Outstanding incoming and outgoing message limits

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.
- Login, remote login, and saveabend flags
- File creation mask

If the *pe_parms* parameter contains a null pointer, the default Guardian attributes of the new process that differ from those of the calling process are:

- Segments created or shared using Guardian procedures such as SEGMENT_ALLOCATE_ are not inherited.
- The program file is the file specified in the **tdm_spawn( )** call.
- The library file is specified in the program file.
- The child process does not inherit the parent process extended swap file (if any). For a G-series TNS process or an accelerated process, the extended data segment is managed by the Kernel Managed Storage Facility (KMSF) unless an extended swap file is specified in the **pe_extwap_file_name** field of the **process_extension** structure described elsewhere in this reference page.

- The process name for the new process is system-generated if the RUNNAMED option is set in the program file. Otherwise, the process is unnamed.
- The size of the data segment of the new process is set in the program file.
- The remote login flag (PCBREMID) is set to zero (off) if the program file has its **S_ISUID** mode bit set. Otherwise, the remote login flag is set the same as for the caller.
- The size of the extended data segment of the new process is set in the program file.
- The DEFINEs inherited by the new process depend on the setting of DEFINE mode in the caller. If DEFINE mode in the caller is ON, all the caller's DEFINEs are inherited. If DEFINE mode is OFF, no DEFINEs are inherited.
- The process identification number (PIN) of the child process is unrelated to that of the parent process. Usually, the PIN of the child process is unrestricted. However, the PIN can be restricted to the range 0 through 254 under the following conditions:
  - — The HIGHPIN flag is not set in, or is absent from, the program file or any library file.
  - — **_TPC_HIGHPIN_OFF** is specified in the **pe_create_options** field of the **process_extension** structure, described following.
  - — The restriction is inherited. See **_TPC_IGNORE_FORCEPIN_ATTR** in the **pe_create_options** field of the **process_extension** structure, described following, for more information about controlling inheritance.
- The process access ID (PAID) depends on whether the **S_ISUID** mode bit of the process image file is set. If that bit is set, the PAID is based on the file owner ID. If not, the PAID is the same as for the caller. (The **S_ISUID** mode bit of the image file has no effect on the security group list.)
- For unnamed processes, the MOM field of the child process is NULL. For named processes, the ancestor field identifies the parent.
- System debugger selection for the new process is based on the INSPECT mode of the program file.
- Code breakpoints and memory breakpoints are not inherited.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

**Setting Guardian Attributes**

The input structure pointed to by the *pe_parms* parameter permits the setting of Guardian attributes for the new process.

First, the input structure must be initialized to the default values (see **Default Guardian Attributes**, earlier in this reference page) using the **#define DEFAULT_PROCESS_EXTENSION**. After the structure is initialized, the values can be set using literals that are defined in the **tdmext.h** header file.

If any optional parameter specified in the structure pointed to by *pe_parms* is not passed, the new process assumes the corresponding default value.

The input structure is defined in the **tdmext.h** header file. This structure can contain fields that vary from release version update (RVU) to RVU, including reserved and filler fields.

In the current RVU, these fields are meaningful:

```
#if defined (__LP64) || defined (_PROCEX32_64BIT)
   typedef struct process_extension {
           short    pe_ver;
           short    pe_len;
           int      pe_pfs_size;
           long long    pe_mainstack_max;
           long long    pe_heap_max;
           long long    pe_space_guarantee;
           char _ptr64  *pe_library_name;
           char _ptr64  *pe_swap_file_name;
           char _ptr64  *pe_extswap_file_name;
           char _ptr64  *pe_process_name;
           char _ptr64  *pe_hometerm;
           char _ptr64  *pe_defines;
           short    pe_defines_len;
           short    pe_priority;
           short    pe_cpu;
           short    pe_memory_pages;
           short    pe_jobid;
           short    pe_name_options;
           short    pe_create_options;
           short    pe_debug_options;
           short    pe_OSS_options;
           char     filler_1[6];
   } process_extension_def;
#else /* !defined (__LP64) && !defined (_PROCEX32_64BIT) */
   typedef struct process_extension {
           long     pe_len;
           char     *pe_library_name;
           char     *pe_swap_file_name;
           char     *pe_extswap_file_name;
           short    pe_priority;
           short    pe_cpu;
           short    pe_name_options;
           char     filler_1[2];
           char     *pe_process_name;
           char     *pe_hometerm;
           short    pe_memory_pages;
           short    pe_jobid;
           short    pe_create_options;
           char     filler_2[2];
           char     *pe_defines;
           short    pe_defines_len;
           short    pe_debug_options;
           long     pe_pfs_size;
           short    pe_OSS_options;
           char     filler_3[2];
           long     pe_mainstack_max;
           long     pe_heap_max;
           long     pe_space_guarantee;
   } process_extension_def;
```

**#endif /* !defined (__LP64) && !defined (_PROCEX32_64BIT) */**

When an application is compiled in 64-bit compile mode or compiled using the **#define _PROCEX32_64BIT 1** feature test macro or an equivalent compiler command option, the application will use the version of the **process_extension** structure that contains 64-bit data types. The **_PROCEX32_64BIT** flag is only required if a 32-bit process must specify larger 64-bit values for **pe_mainstack_max**, **pe_heap_max**, and **pe_space_guaranter**. These larger data types are optional when creating a 64-bit process.

**Note:** The input structure supports two versions: one that contains 64-bit data types and one that contains 32-bit data types. Because the order in which the fields appear in this structure varies significantly based on the version in use, the field definitions below are defined alphabetically instead of sequentially.

The input structure passes this information:

**pe_cpu** Specifies the processor on which the new process will execute. The OSS process ID (PID) of the process remains unchanged. This field is used to distribute system load.

**pe_create_options**
Specifies process creation options as:

**_TPC_BOTH_DEFINES**
Propagates the current DEFINEs and the DEFINEs indicated in the input structure.

**_TPC_ENABLE_DEFINES**
Enables DEFINEs when set if **_TPC_OVERRIDE_DEFMODE** is also set. Disables DEFINEs when not set.

**_TPC_HIGHPIN_OFF**
Restricts the new process to a PIN in the range 0 through 254. This restriction is rarely useful for an OSS process; it allows obsolescent Guardian interfaces to interact with the process.

By default, this restriction is inherited by any child or successor process. The default can be overridden by using the **_TPC_IGNORE_FORCEPIN_ATTR** field.

**_TPC_IGNORE_FORCEPIN_ATTR**
Ignores the **_TPC_HIGHPIN_OFF** restriction specified for or inherited by the caller or parent process. When **_TPC_IGNORE_FORCEPIN_ATTR** is specified, the resulting process has a restricted PIN only if **_TPC_HIGHPIN_OFF** is also specified or if the object file for the program or a user library lacks the HIGHPIN attribute.

**_TPC_OVERRIDE_DEFMODE**
Specifies that the DEFINE mode of the new process is to be set according to the **_TPC_ENABLE_DEFINES** option rather than to the caller’s current DEFINE mode.

**_TPC_PROCESS_DEFINES_ONLY**
Propagates only the current set of DEFINEs.

**_TPC_SUPPLIED_DEFINES_ONLY**
Propagates only the DEFINEs indicated by the **pe_defines** field.

**pe_debug_options**
Provides control over the selection between the default and symbolic debuggers and over the creation of the saveabend file. A saveabend file can be examined by using the symbolic debugger to determine the cause of the abnormal termination. In addition, you can use this option to force the new process to enter the default debugger before executing. Possible options are:

**_TPC_CODEFILE_INSPECT_SAVEABEND**
Uses the saveabend and INSPECT mode flags in the program file.

**_TPC_DEBUG_NOSAVE**
Uses the default debugger but does not create a saveabend file.

**_TPC_DEBUG_SAVEABEND**
Uses the default debugger and creates a saveabend file.

**_TPC_ENTER_DEBUG**
Starts the new process in the default debugging utility.

**_TPC_INSPECT_NOSAVE**
Uses the symbolic debugger but does not create a saveabend file.

**_TPC_INSPECT_SAVEABEND**
Uses the symbolic debugger and creates a saveabend file.

**pe_defines** Points to a specified saved set of DEFINEs created by using the Guardian DEFINESAVE procedure. These DEFINEs are propagated to the new process if either **_TPC_SUPPLIED_DEFINES_ONLY** or **_TPC_BOTH_DEFINES** is specified in the **pe_create_options** field.

**Note:** This string is not null-terminated.

**pe_defines_len**
Specifies the length of the string in the **pe_defines** field.

**pe_extswap_file_name**
Points to a null-terminated string specifying the name of a disk file in the Guardian file system to be used as the swap file for the extended data segment. For example, if the Guardian filename is $A.B.D, the name used is **/G/a/b/d**.

This file cannot have the same name as that of a file used in a preceding call to the **tdm_fork( )** function.

This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is checked for validity but otherwise ignored.

By default, the new process uses KMSF to manage its extended swap segment. HP recommends using the default.

**pe_heap_max** Specifies the maximum size of the heap in bytes for the new process if it is a native process.

See the *C/C++ Programmer's Guide* description of the **HEAP** pragma for guidance on the use of nonzero values for this field.

If a value is specified for this field for G-series TNS or accelerated object files, the specified value is ignored.

**pe_hometerm** Points to the null-terminated name in the Guardian file system for the home terminal. For example, if the Guardian name is $ztnt.#xyz, the name used is **/G/ztnt/#xyz**.

**pe_jobid** Specifies the job ID of the new process.

**pe_len** Specifies the size of the structure in bytes. This value is set by **#define DEFAULT_PROCESS_EXTENSION** and should not be changed.

**pe_library_name**

Points to the name of the user library to be bound to the new process. The string that is pointed to is null-terminated and in OSS name format. If the pointer points to a zero-length string (a NULL character), the new process runs with no user library. An equivalent call to the Guardian PROCESS_LAUNCH_ procedure does this by setting the library filename length to -1.

This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is ignored.

**pe_mainstack_max**

Specifies the maximum size of the main stack in bytes for the new process.

If the calling process specifies a value, the value must be less than 32 MB. If the calling process does not specify a value or specifies a 0 (zero) value, the value specified in the object file of the new process is used. If no value is specified in the object file, the default value of 1 MB (for TNS/R systems) or 2 MB (for TNS/E systems) is used.

**pe_memory_pages**

Specifies the size of the data stack in 2 KB units. This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is checked for validity but otherwise ignored.

**pe_name_options**

Specifies process naming as:

**_TPC_GENERATE_NAME**

The system generates the name.

**_TPC_NAME_SUPPLIED**

The process name is indicated by the **pe_process_name** field.

**_TPC_NO_NAME**

The new process is unnamed.

**pe_OSS_options**

Specifies OSS options. No special action on signals is the default and only current OSS option.

**pe_pfs_size** Specifies the size of the PFS for the new process (this field is ignored).

**pe_priority** Specifies the priority of the new process.

**pe_process_name**
Points to the null-terminated Guardian process name if **_TPC_NAME_SUPPLIED** is specified in the **pe_name_options** field. For example, if the Guardian process name is $DELM, the name used is **/G/delm**.

**pe_space_guarantee**
Specifies the minimum available swap space to guarantee for the new process.

If the calling process specifies a value, the value must be less than or equal to a multiple of the page size of the processor in which the new process will run. Values less than a multiple of the page size are rounded up to the next multiple of the page size. If the calling process does not specify a value or specifies a 0 (zero) value, the value specified in the native object file of the new process is used. If no value is specified in the native object file, the default value of 0 (zero) is used, and enough swap space is guaranteed to launch the process.

If the new process requires a guarantee of available swap space and the system cannot guarantee the required amount, the function call fails, and **errno** is set to the value of [EAGAIN].

If a value is specified for this field for G-series TNS or accelerated object files, the specified value is used for the main stack of the new process.

**pe_swap_file_name**
Points to a null-terminated string specifying the name of a file in the Guardian file system to be used as the swap file for the stack segment. For example, if the Guardian filename is $A.B.C, the name used is **/G/a/b/c**.

This file cannot have the same name as that of a file used in a preceding call to the **tdm_fork( )** function.

This field is not used in the current RVU of Open System Services. It exists for compatibility with older RVUs. Any specified value is checked for validity but otherwise ignored.

**pe_ver** Specifies the version of the **process_extension** structure. This value is set by **#define DEFAULT_PROCESS_EXTENSION** and should not be changed.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Output Structure Information

If the *pr_results* parameter does not contain a null pointer, it points to an output structure defined in the **tdmext.h** header file. This structure can contain fields that vary from RVU to RVU, including reserved and filler fields.

First, the output structure must be initialized by using the **#define DEFAULT_PROCESS_EXTENSION_RESULTS**. This initialization sets the value of the **pr_len** field to the correct value for the current RVU. The value of the **pr_len** field should not be modified after being set by **#define DEFAULT_PROCESS_EXTENSION_RESULTS**.

The **process_extension_results** output structure is described in the **process_extension_results(5)** reference page.

## EXAMPLES

This example uses the **tdm_spawn( )** function to perform I/O redirection in a new process:

```
if ((NewStdOut = open ("newout", ...)) != -1)
        /* process the error */
```

fd_map[0] = 0;
fd_map[1] = NewStdOut;
fd_map[2] = 2;
fd_count = 3;
tdm_spawn(..., fd_count, fd_map, ...);
close(NewStdOut);

This example creates a new process under a different user ID:

```
save = getuid();
setuid(newid);
tdm_spawn(...);
setuid(save);
```

## RETURN VALUES

Upon successful completion, the **tdm_spawn( )** function returns the OSS process ID of the child process to the parent process. If the *pr_results* parameter does not contain a null pointer, it returns the Guardian process handle of the new process in addition to the OSS process ID.

If the **tdm_spawn( )** function fails, the value **-**1 is returned to the parent process, no child process is created, and **errno** is set to indicate the error. If the *pr_results* parameter does not contain a null pointer, the structure it points to returns additional error information, including the PROCESS_LAUNCH_ error and error detail.

## ERRORS

If any of the following conditions occurs, the **tdm_spawn( )** function sets **errno** to the corresponding value, file descriptors marked close-on-exec are not closed, signals set to be caught are not set to the default action, and none of these are changed:

- The *argv*[ ] array of pointers
- The *envp*[ ] array of pointers
- The elements pointed to by these arrays
- The value of the global variable **environ**
- The pointers contained within the global variable **environ**
- The elements pointed to by **environ** pointers
- The effective user ID of the current process
- The effective group ID of the current process

[E2BIG] The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit. The limit can be obtained by calling the **sysconf(_SC_ARG_MAX)** function.

[EACCES] One of these conditions exists:

- Search permission is denied for the directory components of the pathname prefix to the process image file.

- The new process image file, any library file, or script file denies execution permission.
- Create access on the extended swap file on a disk under Safeguard protection is denied.

  This error occurs only for G-series TNS or accelerated new process image files.
- The new process image file is not a regular file.

[EAGAIN] System resources such as disk space, process control block (PCB) space, MAPPOOL space, stack space, or PFS space are temporarily inadequate.

[EBADF] A file descriptor pointed to by the *fd_map***[ ]** parameter is invalid.

[EFAULT] An address for a parameter in the **process_extension** structure pointed to by *pe_parms* is out of allowable bounds. The Guardian PROCESS_LAUNCH_ error and error detail information is returned in the structure pointed to by the *pr_results* parameter, unless *pr_results* contains a null pointer.

[EHLDSEM] The process tried to create a child process in a different processor while having at least one **semadj** value.

[EINVAL] One of these conditions exists:

- An invalid parameter value was supplied in the **process_extension** structure pointed to by *pe_parms*. The Guardian PROCESS_LAUNCH_ error and error detail information is returned in the structure pointed to by the *pr_results* parameter, unless *pr_results* contains a null pointer.
- The new process image file is a binary executable file with invalid attributes.

[EIO] Some physical input or output error has occurred. Either a file cannot be opened because of an input or output error or data has been lost during an input or output transfer. This value is used only for errors on the object file of a loaded program or library, or during data transfer with a Guardian environment home terminal.

For systems running J06.07 and later J-series RVUs or H06.18 or later H-series RVUs, this error can also occur when the OSS file system is out of memory and one or more open files cannot be propagated from the parent process to the child process. In this case, if you are running a program from the shell with the shell reporting any errors, you might see an error like this:

/bin/-sh: /bin/ps: tdm_execve(): failed with unexpected error pr_errno=(4005) pr_TPCerror=(110) pr_TPCdetail=(36)

where:

- **pr_errno** is the [EIO] error
- **pr_TPCerror** is the Guardian PROCESS_LAUNCH_ or PROCESS_CREATE_ error.

[ELOOP] Too many symbolic links were encountered in pathname resolution.

[EMFILE] The maximum number of files are open. The process attempted to open more than the maximum number of file descriptors allowed for the process. One of these conditions might exist:

- The maximum value for *fd_count* has been exceeded.
- The process file segment (PFS) of the child process is smaller than that of the parent process.

[ENAMETOOLONG]
One of these is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the pathname pointed to by the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOCPU] The selected processor does not exist, or the selected processor is down or otherwise unavailable for process creation.

[ENODEV] The system cannot find the fileset containing the process image file.

[ENOENT] One of these conditions exists:

- One or more components of the new process image file's pathname do not exist.
- The *path* parameter points to an empty string.

[ENOEXEC] The new process image file has the appropriate access permissions and is in the OSS name space, but it is neither in the correct binary executable format nor a valid executable text file.

[ENOMEM] Required resources are not available. Subsequent calls to the same function will not succeed for the same reason.

Possible causes of this error include insufficient primary memory (stack, globals, or heap) for the new process.

[ENOTDIR] A component of the path prefix of the new process image file is not a directory.

[ENOTOSS] The calling process is not an OSS process. The **tdm_spawn( )** function cannot be used from the Guardian environment.

[EPERM] One of the following conditions exist:

- The value of the *inherit*->**pgroup** field does not match any process group ID in the same session as the calling process.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[ETXTBSY] The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.

[EUNKNOWN]
Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

The structure pointed to by the *pr_results* parameter might contain additional Guardian PROCESS_LAUNCH_ procedure error and error detail information if any of these errors occur: [EACCES], [EAGAIN], [EFAULT], [EINVAL], [EIO], [ENOCPU], and [ENOEXEC].

## RELATED INFORMATION

Commands: **eld(1)**, **ld(1)**, **nld(1)**.

Functions: **alarm(3)**, **chmod(2)**, **exec(2)**, **_exit(2)**, **exit(3)**, **fcntl(2)**, **fork(2)**, **getenv(3)**, **putenv(3)**, **semget(2)**, **shmat(2)**, **sigaction(2)**, **system(3)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawnp(2)**, **times(3)**, **ulimit(2)**, **umask(2)**.

Files: **signal(4)**.

Miscellaneous: **environ(5)**, **process_extension_results(5)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification.

## NAME

**tdm_spawnp** - Executes a new process with HP extensions

## LIBRARY

G-series native OSS processes: **/G/system/sys*nn*/zossksrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/zosskdll**
64-bit H-series and J-series OSS processes: **/G/system/zdll*nnn*/yosskdll**

## SYNOPSIS

**#include <spawn.h>**
**#include <tdmext.h>**

[ **extern char **environ;** ]

**pid_t tdm_spawnp(**
  **const char** **file*,
  **const int** *fd_count*,
  **const int** *fd_map*[ ],
  **const struct inheritance** **inherit*,
  **char * const** *argv*[ ],
  **char * const** *envp*[ ],
  **const struct process_extension** **pe_parms*,
  **struct process_extension_results** **pr_results*);

## PARAMETERS

****environ**
: Points to an array of character pointers to environment strings. The environment strings define the OSS environment for the parent process. The **environ** array is terminated by a null pointer.

*file*
: Points to a pathname that identifies the new process image file. If the pathname:

- Starts with a slash (/) character; it is the absolute pathname.
- Does not start with a slash but does contain a slash; the pathname resolves relative to the current working directory.
- Contains no slash, the system searches the directories listed in the **PATH** environment variable for the file and prefixes the directory in which the file is found.

*fd_count*
: Specifies the number of file descriptors designated by the *fd_map*[ ] parameter. All file descriptors higher than *fd_count* are closed in the child process. This parameter can take values from 0 (zero) through **POSIX_OPEN_MAX**.

*fd_map*[ ]
: Maps file descriptors from the parent process to the child process. File descriptors identified with the value **SPAWN_FDCLOSED** are closed in the child process.

If this parameter is a null pointer, all open OSS file descriptors of the parent process (except for files opened by Guardian function or procedure calls and those with the **FD_CLOEXEC** attribute flag set) are inherited by the child process. Such inherited file descriptors behave here as they do for the **tdm_execvep( )** function.

*inherit* Points to a structure that allows the process group ID and signal mask of the new process to be specified in addition to a list of signals that the child process will take default action on. The structure is defined in the **spawn.h** header file.

*argv***[ ]** Specifies an array of character pointers to null-terminated strings containing arguments to be passed to the main function of the new program. *argv***[0]** should point to the null-terminated string containing the filename of the new process image. The last member of this array must be a null pointer.

*envp***[ ]** Specifies an array of character pointers to null-terminated strings that describe the environment for the new process.

*pe_parms* Points to the input structure containing Guardian process attributes to be assigned to the new process. The structure is defined in the **tdmext.h** header file.

When this parameter contains a null pointer, the **tdm_spawnp( )** function assumes default Guardian attributes. Otherwise, the structure must be defined locally and initialized before its first use. Initialization is done using the **#define DEFAULT_PROCESS_EXTENSION**, as defined in the **tdmext.h** header file. The initialized values can then be modified as appropriate for the call.

*pr_results* Points to the output structure containing optional process identification and error information. In case of error, this structure provides additional information, including the PROCESS_LAUNCH_ procedure error and error detail. The structure is defined in the **tdmext.h** header file.

The structure must be defined locally and initialized before its first use. Initialization is done by using the **#define DEFAULT_PROCESS_EXTENSION_RESULTS**, as defined in the **tdmext.h** header file.

See the **process_extension_results(5)** reference page for information about the content of the structure. The **tdmext.h** header file is not kept current when new error codes are defined for process creation functions. The list of **_TPC_** macros described in that reference page is not complete; for a current description of error macros and error codes, see the Guardian header file $SYSTEM.ZSPIDEF.ZGRDC or the summary of process-creation errors in the *Guardian Procedure Calls Reference Manual* (see the table entitled "Summary of Process Creation Errors").

## DESCRIPTION

The **tdm_spawnp( )** function creates a new process image. The new image is constructed from a regular executable file, called a new process image file. The new process image file is formatted as an executable text or binary file in one of the formats recognized by the **tdm_spawnp( )** function.

The **tdm_spawnp( )** function is similar to the **tdm_spawn( )** function. The main difference is the way the pathname for the process image file is resolved. **tdm_spawn( )** always resolves relative pathnames by prefixing the current working directory. **tdm_spawnp( )** sometimes uses the **PATH** environment variable to resolve pathnames; see **Identifying the Process Image File**, later in this reference page.

The **tdm_spawnp( )** function provides a different way to create a new process than the way provided by the **tdm_fork( )** and **tdm_execvep( )** functions. **tdm_spawnp( )** provides a more efficient way to create a new process to execute a new program file. However, **tdm_spawnp( )** does not provide all the function provided by **tdm_fork( )** and **tdm_execvep( )**.

When a program is executed as a result of a **tdm_spawnp( )** call, it is entered as a function call:

**int main(**
**int** *argc***,**
**char** **argv***[ ],**
**char** **envp***);**

Here, the *argc* parameter is the argument count, the *argv***[ ]** parameter is an array of character pointers to the arguments themselves, and the *envp* parameter is a pointer to a character array listing the environment variables. The *argv***[ ]** array is terminated by a null pointer. The null pointer is not counted in *argc*.

The arguments specified by a program using the **tdm_spawnp( )** function are passed on to the new process image in the corresponding arguments to the **main( )** function.

**Use From the Guardian Environment**

If called from a Guardian process, the actions of this function are undefined, and **errno** is set to [ENOTOSS].

**Identifying the Process Image File**

The **tdm_spawnp( )** function uses the *file* parameter to identify the process image file. If the pathname pointed to by the *file* parameter starts with a slash ( / ) character, it is the absolute pathname. If the pathname does not start with a slash but contains a slash, the pathname is resolved relative to the current working directory. Otherwise, the pathname does not contain a slash; the system searches the directories listed in the **PATH** environment variable for the file and prefixes the directory in which the file is found.

**Passing the Arguments**

The *argv***[ ]** parameter is an array of character pointers to null-terminated strings. The last member of this array is a null pointer. These strings constitute the argument list available to the new process image. The value in *argv***[0]** should point to a filename that is associated with the process being started by the **tdm_spawnp( )** function.

**Specifying the Environment**

The *envp***[ ]** parameter is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated with a null pointer.

The number of bytes available for the new process's combined argument and environment lists has a system-imposed limit. This limit, which includes the pointers and the null terminators on the strings, is available by calling the **sysconf(_SC_ARG_MAX)** function.

**Executing a Binary File**

If the file specified as the new process image file is a binary executable file, the **tdm_spawnp( )** function loads the file directly.

**Executing a Text File**

If the file specified as the new process image file is not a binary executable file, the **tdm_spawnp( )** function examines the file to determine whether it is an executable text file. It checks for a header line in this format:

**#!** *interpreter_name* [*optional_string*]

The **#!** notation identifies the file as an executable text file. The new process image filename is constructed from the process image filename in the *interpreter_name* string, treating it like the *file* parameter. The Guardian input and output structures pointed to by the *pe_parms* and *pr_results* parameters apply to the command interpreter as they would to any process file.

The arguments passed to the new process are modified as listed:

- *argv***[0]** is set to the name of the command interpreter.
- If the *optional_string* portion is present, *argv***[1]** is set to *optional_string*.
- The next element of *argv***[ ]** is set to the original value of *file*.
- The remaining elements of *argv***[ ]** are set to the original elements of *argv***[ ]**, starting with *argv***[1]**. The original *argv***[0]** is discarded.

The **S_ISUID** and **S_ISGID** mode bits of an executable text file are ignored.

**When the File Is Invalid**

If the process image file is not a valid executable object, and it is a regular text file that does not contain the header line, the **tdm_spawnp( )** function invokes the *interpreter_name* command interpreter as the new process image and passes these arguments to it:

- *argv***[0]** is set to the string "sh".
- *argv***[1]** is set to the original value of the *file* parameter.
- The remaining elements of *argv***[ ]** are set to the original elements of *argv***[ ]** starting with *argv***[1]**.
- The original *argv***[0]** is discarded.

**Open Files**

The *fd_count* and *fd_map***[ ]** parameters determine which file descriptors that were open in the calling process remain open in the child process.

*fd_count* specifies the number of file descriptors to be designated by the *fd_map***[ ]** parameter.

*fd_map***[ ]** specifies how file descriptors in the parent process map to file descriptors in the child process. That is, the file descriptor in *fd_map***[0]** is copied to file descriptor 0 (zero) in the child process, the file descriptor in *fd_map***[1]** is copied to file descriptor 1 in the child process, and so on. If *fd_map***[ ]** has a null value, the *fd_count* parameter is ignored and all open file descriptors in the parent (except for files opened by Guardian function or procedure calls and those with the **FD_CLOEXEC** attribute flag set) are inherited without mapping by the child process. Such inherited file descriptors behave here as they do for the **tdm_execvep( )** function.

If *fd_map***[ ]** does not have a null value, file descriptors from *fd_count* to **OPEN_MAX** are closed in the child process, as are entries in *fd_map***[ ]** that are identified with the value **SPAWN_FDCLOSED**.

If a file descriptor specified in *fd_map***[ ]** is invalid, the function call fails. (Any file descriptor created by a Guardian function or procedure call is invalid.) The **errno** variable is set to [EBADF].

For a G-series TNS process image or an accelerated process image, if the process file segment (PFS) of the new process image is smaller than the process file segment of the calling process image and if the calling process image has a large number of file descriptors open, then the system might not be able to propagate all the open file descriptors to the new process image. When this situation occurs, the function call fails, and **errno** is set to the value of [EMFILE].

**Open Pipes and FIFOs**

A pipe or FIFO associated with an open file descriptor in the parent process remains connected in the child process. If the child process runs in a different processor than the parent process, the processor that runs the child process must also be running an OSS pipe server process.

If no OSS pipe server process is running in the new processor, the child process cannot use the pipe or FIFO; calls specifying the file descriptor for the pipe or FIFO fail with **errno** set to [EWRONGID]. The child process can only close the invalid file descriptor.

**Existing Sockets**

A socket associated with an open file descriptor in the calling process remains connected in the new process when the new process runs in the same processor as the calling process.

When the new process runs in a different processor than the calling process, the processor that runs the new process must also be running a socket transport agent process. If no socket transport agent process is running in the new processor, the new process cannot use the socket; calls specifying the file descriptor for the socket fail with **errno** set to [EWRONGID]. The new process can only close the invalid file descriptor.

**Sharing Guardian Files**

After a successful call to the **tdm_spawnp( )** function, the initial position within an open EDIT file (file code 101) in the Guardian file system (a file in **/G**) that was opened by a call to the OSS **open( )** function is the same for both the parent and child processes. However, the position is not shared; that is, changing the position used by one process does not change the position used by the other process.

**Shared Memory**

Any attached shared memory segments are detached from the child process by a successful call to the **tdm_spawnp( )** function. See the **shmat(2)** reference page for additional information about shared memory segment use.

**Semaphores**

Semaphore set IDs attached to a parent process are also attached to the child process if the child process executes in the same processor as the parent.

A semaphore set cannot be shared when a **semadj** value exists for the parent process and the child process is created in a different processor. When that condition exists, a call to the **tdm_spawnp( )** function fails and **errno** is set to [EHLDSEM].

See the **semget(2)** reference page for additional information about semaphore use.

**Signals**

The setting of signaling attributes in the new process depends on the information provided in the **inheritance** structure (pointed to by the *inherit* parameter).

This default signal information applies to the child process unless modified by the information in the **inheritance** structure:

- Signals set to the default action (**SIG_DFL**) in the parent process are set to the default action in the child process.
- Signals set to be ignored (**SIG_IGN**) by the parent process are set to be ignored by the child process.
- Signals that cause abnormal termination (**SIG_ABORT**) in the calling process image are set to that action in the new process image.
- Signals that cause entry into the debugger (**SIG_DEBUG**) in the calling process image are set to that action in the new process image.
- Signals set to be caught by the parent process are set to the default action in the child process (see the **signal(4)** reference page).
- The signal mask in the child process is inherited from the parent process.

- Signals pending in the parent process are disregarded by the child process.

The **inheritance** structure can modify the default signal information as follows:

- If the **SPAWN_SETSIGMASK** bit is set in *inherit*->**flags**, *inherit*->**sigmask** contains the signal mask for the child process.

- If the **SPAWN_SETSIGDEF** bit is set in *inherit*->**flags**, *inherit*->**sigdefault** specifies the signal set that is forced to the default action in the child process. Additional signals that are set to the default action in the parent process, or for which the parent process has a signal-catching function installed, are also set to the default action in the child process.

**Process Group**

By default, the child process is a member of the same process group as the parent. However, the new process can be designated a member of some other process group by setting the **SPAWN_SETPGROUP** bit in *inherit*->**flags**. The *inherit*->**pgroup** field specifies the process group number, or it contains the **SPAWN_NEWPGROUP** symbolic constant if the new process is to be the leader of a new process group.

**User ID and Group ID**

If the set-user-ID mode bit (**S_ISUID**) of the new process image file is set (see the **chmod(2)** reference page), the effective user ID of the new process image is set to the user ID of the owner of the new process image file. Similarly, if the set-group-ID mode bit (**S_ISGID**) of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved-set user ID and the saved-set group ID) for use by the **setuid( )** function.

**OSS Attributes**

These OSS attributes of the calling process image are unchanged after successful completion of the **tdm_spawnp( )** function:

- Real user ID
- Real group ID
- Session membership
- Current working directory
- Root directory
- File mode creation mask (see the **umask(2)** reference page)
- File size limit (see the **ulimit(2)** reference page)

The OSS attributes of the child process differ from those of the parent process in these ways:

- The child process has a unique OSS process ID (PID) and does not match any active process group ID.

- The parent process ID of the child process matches the OSS process ID of the parent.

- The child process has its own copy of a subset of the parent process's file descriptors. See **Open Files**, earlier in this reference page. However, each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent process.
- The child process does not inherit file opens created by Guardian function or procedure calls.
- The child process does not inherit file locks.
- The child process's **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** values are set to 0 (zero).
- Any pending alarms are cleared in the child process.
- Any adjust-on-exit (**semadj**) values of the parent process are not inherited by the child process.
- Any signals pending for the parent process are not inherited by the child process.
- The signal mask of the child process is that of the parent process unless modified by the *inherit*->**sigmask** field. See **Signals**, earlier in this reference page.
- The set of signals for which default action is set and the set of signals to be ignored are the same in the child process as in the parent process unless modified by *inherit*->**sigdefault**. See **Signals**, earlier.
- The child process does not share directory streams with the parent. All open directory streams are closed for the child process.

**Default Guardian Attributes**

If the *pe_parms* parameter contains a null pointer, the newly created OSS process retains all of these default Guardian attributes of the process that calls the **tdm_spawnp( )** function:

- Priority
- Processor on which the process executes
- Home terminal
- Job ID
- DEFINE mode switch
- Creator access ID (CAID)
- Process access ID (PAID), unless the **S_ISUID** mode bit of the new process image file is set
- Security group list
- Job ancestor or GMOM
- Unread system message index (PCBMCNT)

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.

- Outstanding incoming and outgoing message limits

  This attribute assignment is different from the assignment made when creating a new process with Guardian procedures.

- Login, remote login, and saveabend flags
- File creation mask

If the *pe_parms* parameter contains a null pointer, the default Guardian attributes of the new process that differ from those of the calling process are as follows:

- Segments created or shared using Guardian procedures such as SEGMENT_ALLOCATE_ are not inherited.
- The program file is the file specified in the **tdm_spawnp( )** call.
- The library file is specified in the program file.
- The child process does not inherit the parent process extended swap file (if any). For a G-series TNS process or an accelerated process, the extended data segment is managed by the Kernel Managed Storage Facility (KMSF) unless an extended swap file is specified in the **pe_extwap_file_name** field of the **process_extension** structure described elsewhere in this reference page.
- The process name for the new process is system-generated if the RUNNAMED option is set in the program file. Otherwise, the process is unnamed.
- The size of the data segment of the new process is set in the program file.
- The remote login flag (PCBREMID) is set to zero (off) if the program file has its **S_ISUID** mode bit set. Otherwise, the remote login flag is set the same as for the caller.
- The size of the extended data segment of the new process is set in the program file.
- The DEFINEs inherited by the new process depend on the setting of DEFINE mode in the caller. If DEFINE mode in the caller is ON, all the caller's DEFINEs are inherited. If DEFINE mode is OFF, no DEFINEs are inherited.
- The process identification number (PIN) of the child process is unrelated to that of the parent process. Usually, the PIN of the child process is unrestricted. However, the PIN can be restricted to the range 0 through 254 under the following conditions:

  — The HIGHPIN flag is not set in, or is absent from, the program file or any library file.

  — **_TPC_HIGHPIN_OFF** is specified in the **pe_create_options** field of the **process_extension** structure, described elsewhere in this reference page.

  — The restriction is inherited. See **_TPC_IGNORE_FORCEPIN_ATTR** in the **pe_create_options** field of the **process_extension** structure, described elsewhere in this reference page, for more information about controlling inheritance.

- The process access ID (PAID) depends on whether the **S_ISUID** mode bit of the process image file is set. If that bit is set, the PAID is based on the file owner ID. If not, the PAID is the same as for the caller. (The **S_ISUID** mode bit of the image file has no effect on the security group list.)

- For unnamed processes, the MOM field of the child process is NULL. For named processes, the ancestor field identifies the parent.
- System debugger selection for the new process is based on the INSPECT mode flag in the program file.
- Code breakpoints and memory breakpoints are not inherited.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

**Setting Guardian Attributes**

The input structure pointed to by the *pe_parms* parameter permits the setting of Guardian attributes for the new process.

First, the input structure must be initialized to the default values (see **Default Guardian Attributes**, earlier in this reference page) using the **#define DEFAULT_PROCESS_EXTENSION**. After the structure is initialized, the values can be set using literals that are defined in the **tdmext.h** header file.

If any optional parameter specified in the structure pointed to by *pe_parms* is not passed, the new process assumes the corresponding default value.

The input structure is defined in the **tdmext.h** header file. This structure can contain fields that vary from release version update (RVU) to RVU, including reserved and filler fields.

In the current RVU, these fields are meaningful:

```
#if defined (__LP64) || defined (_PROCEX32_64BIT)

   typedef struct process_extension {
          short    pe_ver;
          short    pe_len;
          int      pe_pfs_size;
          long long    pe_mainstack_max;
          long long    pe_heap_max;
          long long    pe_space_guarantee;
          char _ptr64  *pe_library_name;
          char _ptr64  *pe_swap_file_name;
          char _ptr64  *pe_extswap_file_name;
          char _ptr64  *pe_process_name;
          char _ptr64  *pe_hometerm;
          char _ptr64  *pe_defines;
          short    pe_defines_len;
          short    pe_priority;
          short    pe_cpu;
          short    pe_memory_pages;
          short    pe_jobid;
          short    pe_name_options;
          short    pe_create_options;
          short    pe_debug_options;
          short    pe_OSS_options;
          char     filler_1[6];
   } process_extension_def;

#else  /* !defined (__LP64) && !defined (_PROCEX32_64BIT) */

   typedef struct process_extension {
          long     pe_len;
          char     *pe_library_name;
```

```
char     *pe_swap_file_name;
         char     *pe_extswap_file_name;
         short    pe_priority;
         short    pe_cpu;
         short    pe_name_options;
         char     filler_1[2];
         char     *pe_process_name;
         char     *pe_hometerm;
         short    pe_memory_pages;
         short    pe_jobid;
         short    pe_create_options;
         char     filler_2[2];
         char     *pe_defines;
         short    pe_defines_len;
         short    pe_debug_options;
         long     pe_pfs_size;
         short    pe_OSS_options;
         char     filler_3[2];
         long     pe_mainstack_max;
         long     pe_heap_max;
         long     pe_space_guarantee;
} process_extension_def;
#endif /* !defined (__LP64) && !defined (_PROCEX32_64BIT) */
```

When an application is compiled in 64-bit compile mode or compiled using the **#define _PROCEX32_64BIT 1** feature test macro or an equivalent compiler command option, the application will use the version of the **process_extension** structure that contains 64-bit data types. The **_PROCEX32_64BIT** flag is only required if a 32-bit process must specify larger 64-bit values for **pe_mainstack_max**, **pe_heap_max**, and **pe_space_guaranter**. These larger data types are optional when creating a 64-bit process.

**Note:** The input structure supports two versions: one that contains 64-bit data types and one that contains 32-bit data types. Because the order in which the fields appear in this structure varies significantly based on the version in use, the field definitions below are defined alphabetically instead of sequentially.

The input structure passes this information:

**pe_cpu**
Specifies the processor on which the new process will execute. The OSS process ID (PID) of the process remains unchanged. This field is used to distribute system load.

**pe_create_options**
Specifies process creation options as:

**_TPC_BOTH_DEFINES**
Propagates the current DEFINEs and the DEFINEs indicated in the input structure.

**_TPC_ENABLE_DEFINES**
Enables DEFINEs when set if **_TPC_OVERRIDE_DEFMODE** is also set. Disables DEFINEs when not set.

**_TPC_HIGHPIN_OFF**
Restricts the new process to a PIN in the range 0 through 254. This restriction is rarely useful for an OSS process; it allows obsolescent Guardian interfaces to interact with the process.

By default, this restriction is inherited by any child or successor process. The default can be overridden by using the **_TPC_IGNORE_FORCEPIN_ATTR** field.

**_TPC_IGNORE_FORCEPIN_ATTR**
Ignores the **_TPC_HIGHPIN_OFF** restriction specified for or inherited by the caller or parent process. When **_TPC_IGNORE_FORCEPIN_ATTR** is specified, the resulting process has a restricted PIN only if **_TPC_HIGHPIN_OFF** is also specified or if the object file for the program or a user library lacks the HIGHPIN attribute.

**_TPC_OVERRIDE_DEFMODE**
Specifies that the DEFINE mode of the new process is to be set according to the **_TPC_ENABLE_DEFINES** option rather than to the caller's current DEFINE mode.

**_TPC_PROCESS_DEFINES_ONLY**
Propagates only the current set of DEFINEs.

**_TPC_SUPPLIED_DEFINES_ONLY**
Propagates only the DEFINEs indicated by the **pe_defines** field.

**pe_debug_options**
Provides control over the selection between the default and symbolic debuggers and over the creation of the saveabend file. A saveabend file can be examined by using the symbolic debugger to determine the cause of the abnormal termination. In addition, you can use this option to force the new process to enter the default debugger before executing. Possible options are:

**_TPC_CODEFILE_INSPECT_SAVEABEND**
Uses the saveabend and INSPECT mode flags in the program file.

**_TPC_DEBUG_NOSAVE**
Uses the default debugger but does not create a saveabend file.

**_TPC_DEBUG_SAVEABEND**
Uses the default debugger and creates a saveabend file.

**_TPC_ENTER_DEBUG**
Starts the new process in the default debugging utility.

**_TPC_INSPECT_NOSAVE**
Uses the symbolic debugger but does not create a saveabend file.

**_TPC_INSPECT_SAVEABEND**
Uses the symbolic debugger and creates a saveabend file.

**pe_defines** Points to a specified saved set of DEFINEs created by using the Guardian DEFINESAVE procedure. These DEFINEs are propagated to the new process if either **_TPC_SUPPLIED_DEFINES_ONLY** or **_TPC_BOTH_DEFINES** is specified in the **pe_create_options** field.

**Note:** This string is not null-terminated.

**pe_defines_len**
Specifies the length of the string in the **pe_defines** field.

**pe_extswap_file_name**
Points to a null-terminated string specifying the name of a disk file in the Guardian file system to be used as the swap file for the extended data segment. For example, if the Guardian filename is $A.B.D, the name used is **/G/a/b/d**.

This file cannot have the same name as that of a file used in a preceding call to the **tdm_fork( )** function.

This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is checked for validity but otherwise ignored.

By default, the new process uses KMSF to manage its extended swap segment. HP recommends using the default.

**pe_heap_max** Specifies the maximum size of the heap in bytes for the new process if it is a native process.

See the *C/C++ Programmer's Guide* description of the **HEAP** pragma for guidance on the use of nonzero values for this field.

If a value is specified for this field for G-series TNS or accelerated object files, the specified value is ignored.

**pe_hometerm** Points to the null-terminated name in the Guardian file system for the home terminal. For example, if the Guardian name is $ztnt.#xyz, the name used is **/G/ztnt/#xyz**.

**pe_jobid** Specifies the job ID of the new process.

**pe_len** Specifies the size of the structure in bytes. This value is set by **#define DEFAULT_PROCESS_EXTENSION** and should not be changed.

**pe_library_name**
Points to the name of the user library to be bound to the new process. The string that is pointed to is null-terminated and in OSS name format. If the pointer points to a zero-length string (a NULL character), the new process runs with no user library. An equivalent call to the Guardian PROCESS_LAUNCH_ procedure does this by setting the library filename length to -1.

This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is ignored.

**pe_mainstack_max**
Specifies the maximum size of the main stack in bytes for the new process.

If the calling process specifies a value, the value must be less than 32 MB. If the calling process does not specify a value or specifies a 0 (zero) value, the value specified in the object file of the new process is used. If no value is specified in the object file, the default value of 1 MB (for TNS/R systems) or 2 MB (for

TNS/E systems) is used.

**pe_memory_pages**
Specifies the size of the data stack in 2 KB units. This field is used only for G-series TNS or accelerated new process image files. If a value is specified for this field for native object files, the specified value is checked for validity but otherwise ignored.

**pe_name_options**
Specifies process naming as:

**_TPC_GENERATE_NAME**
The system generates the name.

**_TPC_NAME_SUPPLIED**
The process name is indicated by the **pe_process_name** field.

**_TPC_NO_NAME**
The new process is unnamed.

**pe_OSS_options**
Specifies OSS options. No special action on signals is the default and only current OSS option.

**pe_pfs_size** Specifies the size of the PFS for the new process (this field is ignored).

**pe_priority** Specifies the priority of the new process.

**pe_process_name**
Points to the null-terminated Guardian process name if **_TPC_NAME_SUPPLIED** is specified in the **pe_name_options** field. For example, if the Guardian process name is $DELM, the name used is **/G/delm**.

**pe_space_guarantee**
Specifies the minimum available swap space to guarantee for the new process.

If the calling process specifies a value, the value must be less than or equal to a multiple of the page size of the processor in which the new process will run. Values less than a multiple of the page size are rounded up to the next multiple of the page size. If the calling process does not specify a value or specifies a 0 (zero) value, the value specified in the native object file of the new process is used. If no value is specified in the native object file, the default value of 0 (zero) is used, and enough swap space is guaranteed to launch the process.

If the new process requires a guarantee of available swap space and the system cannot guarantee the required amount, the function call fails, and **errno** is set to the value of [EAGAIN].

If a value is specified for this field for G-series TNS or accelerated object files, the specified value is used for the main stack of the new process.

**pe_swap_file_name**
Points to a null-terminated string specifying the name of a file in the Guardian file system to be used as the swap file for the stack segment. For example, if the Guardian filename is $A.B.C, the name used is **/G/a/b/c**.

This file cannot have the same name as that of a file used in a preceding call to the **tdm_fork( )** function.

This field is not used in the current RVU of Open System Services. It exists for compatibility with older RVUs. Any specified value is checked for validity but otherwise ignored.

**pe_ver** Specifies the version of the **process_extension** structure. This value is set by **#define DEFAULT_PROCESS_EXTENSION** and should not be changed.

For detailed information about Guardian process attributes, see the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

### Output Structure Information

If the *pr_results* parameter does not contain a null pointer, it points to an output structure defined in the **tdmext.h** header file. This structure can contain fields that vary from RVU to RVU, including reserved and filler fields.

First, the output structure must be initialized by using the **#define DEFAULT_PROCESS_EXTENSION_RESULTS**. This initialization sets the value of the **pr_len** field to the correct value for the current RVU. The value of the **pr_len** field should not be modified after being set by **#define DEFAULT_PROCESS_EXTENSION_RESULTS**.

The **process_extension_results** output structure is described in the **process_extension_results(5)** reference page.

## EXAMPLES

This example uses the **tdm_spawnp( )** function to perform I/O redirection in a new process:

```
if ((NewStdOut = open ("newout", ...)) != -1)
        /* process the error */
```

```
fd_map[0] = 0;
fd_map[1] = NewStdOut;
fd_map[2] = 2;
fd_count = 3;
tdm_spawnp(..., fd_count, fd_map, ...);
close(NewStdOut);
```

This example creates a new process under a different user ID:

```
save = getuid();
setuid(newid);
tdm_spawnp(...);
setuid(save);
```

## RETURN VALUES

Upon successful completion, the **tdm_spawnp( )** function returns the OSS process ID of the child process to the parent process. If the *pr_results* parameter does not contain a null pointer, it returns the Guardian process handle of the new process in addition to the OSS process ID.

If the **tdm_spawnp( )** function fails, the value -1 is returned to the parent process, no child process is created, and **errno** is set to indicate the error. If the *pr_results* parameter does not contain a null pointer, the structure it points to returns additional error information including the PROCESS_LAUNCH_ error and error detail.

**ERRORS**

If any of the following conditions occurs, the **tdm_spawnp( )** function sets **errno** to the corresponding value, file descriptors marked close-on-exec are not closed, signals set to be caught are not set to the default action, and none of these are changed:

- The *argv*[ ] array of pointers
- The *envp*[ ] array of pointers
- The elements pointed to by these arrays
- The value of the global variable **environ**
- The pointers contained within the global variable **environ**
- The elements pointed to by **environ** pointers
- The effective user ID of the current process
- The effective group ID of the current process

[E2BIG] The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit. The limit can be obtained by calling the **sysconf(_SC_ARG_MAX)** function.

[EACCES] One of these conditions exists:

- Search permission is denied for the directory components of the pathname prefix to the process image file.
- The new process image file, any library file, or script file denies execution permission.
- Create access on the extended swap file on a disk under Safeguard protection is denied.

  This error occurs only for G-series TNS or accelerated new process image files.
- The new process image file is not a regular file.

[EAGAIN] System resources such as disk space, process control block (PCB) space, MAPPOOL space, stack space, or PFS space are temporarily inadequate.

[EBADF] A file descriptor pointed to by the *fd_map***[ ]** parameter is invalid.

[EFAULT] An address for a parameter in the **process_extension** structure pointed to by *pe_parms* is out of allowable bounds. The Guardian PROCESS_LAUNCH_ error and error detail information is returned in the structure pointed to by the *pr_results* parameter, unless *pr_results* contains a null pointer.

[EHLDSEM] The process tried to create a child process in a different processor while having at least one **semadj** value.

[EINVAL] One of these conditions exists:

- An invalid parameter value was supplied in the **process_extension** structure pointed to by *pe_parms*. The Guardian PROCESS_LAUNCH_ error and error detail information is returned in the structure pointed to by the *pr_results* parameter, unless *pr_results* contains a null pointer.

- The new process image file is a binary executable file with invalid attributes.

[EIO] Some physical input or output error has occurred. Either a file cannot be opened because of an input or output error or data has been lost during an input or output transfer. This value is used only for errors on the object file of a loaded program or library, or during data transfer with a Guardian environment home terminal.

For systems running J06.07 and later J-series RVUs or H06.18 or later H-series RVUs, this error can also occur when the OSS file system is out of memory and one or more open files cannot be propagated from the parent process to the child process. In this case, if you are running a program from the shell with the shell reporting any errors, you might see an error like this:

/bin/-sh: /bin/ps: tdm_execve(): failed with unexpected error pr_errno=(4005) pr_TPCerror=(110) pr_TPCdetail=(36)

where:

- **pr_errno** is the [EIO] error
- **pr_TPCerror** is the Guardian PROCESS_LAUNCH_ or PROCESS_CREATE_ error.

[ELOOP] Too many symbolic links were encountered in pathname resolution.

[EMFILE] The maximum number of files are open. The process attempted to open more than the maximum number of file descriptors allowed for the process. One of these conditions might exist:

- The maximum value for *fd_count* has been exceeded.
- The process file segment (PFS) of the child process is smaller than that of the parent process.

[ENAMETOOLONG]
One of these is too long:

- The pathname pointed to by the *file* parameter
- A component of the pathname pointed to by the *file* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the value specified by the *file* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOCPU] The selected processor does not exist, or the selected processor is down or otherwise unavailable for process creation.

[ENODEV] The system cannot find the fileset containing the process image file.

[ENOENT] One of these conditions exists:

- One or more components of the new process image file's pathname do not exist.

- The *file* parameter points to an empty string.

[ENOEXEC] The command interpreter could not be invoked following failure to execute the process image file identified by the *file* parameter.

[ENOMEM] Required resources are not available. Subsequent calls to the same function will not succeed for the same reason.

Possible causes of this error include insufficient primary memory (stack, globals, or heap) for the new process.

[ENOTDIR] A component of the path prefix of the new process image file is not a directory.

[ENOTOSS] The calling process is not an OSS process. The **tdm_spawnp( )** function cannot be used from the Guardian environment.

[EPERM] One of the following conditions exist:

- The value of the *inherit*->**pgroup** field does not match any process group ID in the same session as the calling process.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[ETXTBSY] The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.

[EUNKNOWN]
Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

The structure pointed to by the *pr_results* parameter might contain additional Guardian PROCESS_LAUNCH_ procedure error and error detail information if any of these errors occur: [EACCES], [EAGAIN], [EFAULT], [EINVAL], [EIO], [ENOCPU], and [ENOEXEC].

## RELATED INFORMATION

Commands: **eld(1)**, **ld(1)**, **nld(1)**.

Functions: **alarm(3)**, **chmod(2)**, **exec(2)**, **_exit(2)**, **exit(3)**, **fcntl(2)**, **fork(2)**, **getenv(3)**, **putenv(3)**, **semget(2)**, **shmat(2)**, **sigaction(2)**, **system(3)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **times(3)**, **ulimit(2)**, **umask(2)**.

Files: **signal(4)**.

Miscellaneous: **environ(5)**, **process_extension_results(5)**.

## STANDARDS CONFORMANCE

This function is an extension to the XPG4 Version 2 specification.

# Section 9. System Functions (u)

This section contains reference pages for Open System Services (OSS) system function calls with names that begin with **u**. These reference pages reside in the **cat2** directory and are sorted alphabetically by U.S. English conventions in this section.

## NAME

**ulimit -** Sets and gets file size limits

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <ulimit.h>**

```
long int ulimit(
        int cmd [,
        ...
/*              long int blk_size
*/
        ]);
```

In this instance, the ellipsis (. . .) indicates that the function is variable. An additional, optional parameter can be specified.

## PARAMETERS

*cmd* Specifies the operation to be performed. The following values are valid:

**UL_GETFSIZE**
Returns the size limit, in 512-byte blocks, of files opened by the process for writing in the OSS environment. (Files of any size can be read in the OSS environment.)

**UL_SETFSIZE**
Sets the size limit, in 512-byte blocks, of files opened by the process for writing in the OSS environment to the value specified as the second parameter of the call. (Files of any size can be read in the OSS environment.)

This is a restricted operation. Any process can reduce the size limit for its files, but only a process with appropriate privileges can increase the size limit for its files.

*blk_size* Specifies the number of 512-byte blocks to be permitted in a file written by the process. This parameter is required when the *cmd* parameter has the value of **UL_SETFSIZE**. This parameter can be omitted in all other calls.

This parameter must be declared as a **long int** data type.

## DESCRIPTION

The **ulimit( )** function provides control over selected process limits. Limits set by calls to the **ulimit( )** function are inherited by a child process. Limits set by calls to the **ulimit( )** function are enforced only if the file open was created by the OSS **open( )** or **creat( )** function call.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the **ulimit( )** function returns the value of the requested limit. If **ulimit( )** fails, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **ulimit( )** function sets **errno** to the value that corresponds to the condition.

[EINVAL] One of the following conditions exists:

- The value specified for the *cmd* parameter is not valid.
- The value specified for the second parameter is too large.

[EPERM] The process does not have the appropriate privileges to perform the requested operation.

## RELATED INFORMATION

Functions: **creat(2)**, **open(2)**, **write(2)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The error [EINVAL] is returned when the second parameter is too large.

## NAME

**umask -** Sets and gets the value of the file mode creation mask

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <sys/stat.h>

mode_t umask(
        mode_t cmask);
```

## PARAMETERS

*cmask* Specifies the value of the file mode creation mask.

## DESCRIPTION

The **umask( )** function sets the file mode creation mask of the process to the value of the *cmask* parameter and returns the previous value of the mask. The *cmask* parameter is constructed by logically ORing file permission bits defined in the **sys/stat.h** header file.

Whenever a file is created (by the **creat( )**, **mkdir( )**, **mkfifo( )**, **mknod( )**, or **open( )** function), all file permission bits set in the file mode creation mask are cleared in the mode of the created file. This clearing allows users to restrict the default access to their files.

The mask is inherited by child processes.

### Use on Guardian Objects

The file mode creation mask of the process is not used when accessing a file in **/G** (the Guardian file system). If an open causes file creation, the file is given access permissions compatible with the standard security permissions for the Guardian creator access ID (CAID) of the calling process.

During access to a Guardian file, all Guardian environment access permissions are checked. This includes checks by Guardian standard security mechanisms and by the Safeguard product for Guardian disk file and process access.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the previous value of the file mode creation mask is returned.

## RELATED INFORMATION

Commands: **chmod(1)**, **mkdir(1)**, **sh(1)**, **umask(1)**.

Functions: **chmod(2)**, **mkdir(2)**, **mkfifo(3)**, **mknod(2)**, **open(2)**, **stat(2)**.

## NAME

**uname -** Gets information identifying the current system

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/utsname.h>**

**int uname(**
**struct utsname *** *name* **);**

## PARAMETERS

*name* Points to the **utsname** structure, where information about the current system is stored.

## DESCRIPTION

The **uname( )** function stores information identifying the current system in the structure pointed to by the *name* parameter.

The **uname( )** function uses the **utsname** structure, which is defined in the **sys/utsname.h** file as follows:

```
struct utsname {
    char sysname [32];
    char nodename[32];
    char release [8];
    char version [8];
    char machine [16];
};
```

The **uname( )** function returns null-terminated character strings describing the current system.

The **sysname[ ]** array indicates the operating system. For example, the HP implementation uses the value "NONSTOP_KERNEL" on G-series release version updates (RVUs) through at least G06.25.

The **nodename[ ]** array contains the name that the system is known by on an Expand communications network; for example, "boston".

The **release[ ]** array identifies the release version (RV); for example, "H06" might appear for an H-series release version update.

The **version[ ]** array contains the version update number of the RVU. For example, "25" appears for the G06.25 RVU.

The **machine[ ]** array indicates the processor hardware type being used; for example, "NSR-N" or "NSR-T" might be used for a NonStop S-series server, while "NSE-A" might be used for a NonStop Integrity NS-series server.

Because the format and content of the **utsname** structure can change from release to release, it is not advisable to make programmatic choices based on the layout of the fields in this structure.

## RETURN VALUES

Upon successful completion, a nonnegative value is returned. If the function call is unsuccessful, one of the following might happen:

- The value -1 is returned and **errno** is set to indicate the error.
- A Guardian trap is set.

## ERRORS

If the following condition occurs, the **uname( )** function sets **errno** to the corresponding value:

[EFAULT] The *name* parameter points outside of the process address space.

## RELATED INFORMATION

Commands: **uname(1)**.

## STANDARDS CONFORMANCE

The following are HP extensions to the XPG4 Version 2 specification:

- The error [EFAULT] can be returned.

## NAME

**unlink -** Removes a directory entry from the OSS environment

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

**#include <unistd.h>**

**int unlink(**
    **const char** ***path*);**

## PARAMETERS

*path* Specifies the directory entry to be removed.

## DESCRIPTION

The **unlink( )** function removes the directory entry specified by the *path* parameter and decrements the link count of the file referenced by the link.

When all links to a file are removed and no process has the file open, all resources associated with the file are reclaimed and the file is no longer accessible. If one or more processes have the file open when the last link is removed, the link is removed before the **unlink( )** function returns but the removal of the file contents is postponed until all open references to the file are removed. If the *path* parameter names a symbolic link, the symbolic link itself is removed.

The *path* parameter must not name a directory.

The calling process requires both execute (search) and write access permission for the directory containing the file being unlinked. Write permission for an OSS file is not required.

Upon successful completion, the **unlink( )** function marks for update the **st_ctime** and **st_mtime** fields of the directory that contained the entry that was removed. If the file's link count is not 0 (zero) or if the file is open, the **st_ctime** field of the file is also marked for update.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted by the same file permissions and owner privileges as any other user ID: It has no special privileges unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

Executable files that have the PRIVSOARFOPEN privilege and that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use From the Guardian Environment

The **unlink( )** function belongs to a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.

- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

**NOTES**

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

**RETURN VALUES**

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned, the named file is not changed, and **errno** is set to indicate the error.

**ERRORS**

If any of the following conditions occurs, the function sets **errno** to the corresponding value and the named file is not unlinked:

[EACCES] One of the following conditions is true:

- Search permission is denied for a component of the pathname prefix, or write permission is denied on the directory containing the link to be removed.
- The **S_ISVTX flag** is set on the directory containing the existing file referred to by the *path* parameter. However, the calling process is not any of the following:
  - — The file owner
  - — The directory owner
  - — A process with appropriate privileges

[EBUSY] The named file is one of the following:

- The **/dev/tty** file
- The **/dev/null** file

[EFAULT] The *path* parameter is an invalid address.

[EFSBAD] The fileset catalog for one of the filesets involved in the operation is corrupt.

[EGUARDIANOPEN]
One of the following conditions exists:

- The named file is a Guardian file open in the Guardian environment.
- The named file is a Guardian EDIT file (file code 101), and it is open in the OSS environment.

[EINVAL] The named file is a structured file in **/G** (the Guardian file system). Such files cannot be removed by the **unlink( )** function.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of the following conditions exists:

- The named file does not exist.
- The *path* parameter points to an empty string.
- The *path* parameter specifies a file on a remote HP NonStop node but communication with the remote node has been lost.

[ENOROOT] One of the following conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node and communication with the remote name server has been lost.

[ENOTDIR] A component of the pathname prefix is not a directory.

[ENOTSUP] The *path* parameter specifies a Guardian file on an SMF logical volume and one of the following conditions exists:

- The local system is running an RVU prior to J06.15 or H06.26.
- The *path* parameter specifies a file in /E and the remote system is running an RVU prior to J06.15 or H06.26.

[ENXIO] The fileset containing the client's current working directory or root directory is not mounted.

[EOSSNOTRUNNING]
The OSS monitor process is not running.

[EPERM] One of the following conditions exists:

- The named file is a directory.
- The named file is a Guardian file (in **/G**), but it is not a regular file.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The entry to be unlinked is part of a read-only fileset.

[ETXTBSY] One of the following conditions exists:

- The entry to be unlinked is the last directory entry to a file that is already busy.
- The named file is a NonStop SQL/MP object file that is currently executing.

**RELATED INFORMATION**

Commands: **rm(1)**.

Functions: **close(2)**, **link(2)**, **open(2)**, **rmdir(2)**.

**STANDARDS CONFORMANCE**

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- The calling process requires both execute (search) and write access permission for the directory containing the file being unlinked.
- The **unlink( )** function is not supported for directories.

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EFSBAD], [EGUARDIANOPEN], [EINVAL], [ENOROOT], [ENOTSUP], [ENXIO], and [EOSSNOTRUNNING] can be returned.

## NAME

**utime** - Sets file access and modification times

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <utime.h>

int utime(
        const char *path,
        struct utimbuf *times);
```

## PARAMETERS

*path* Points to the pathname for the file. If the final component of the *path* parameter names a symbolic link, the link is traversed and pathname resolution continues.

*times* Points to a **utimbuf** structure containing time values for the file.

## DESCRIPTION

The **utime( )** function sets the access and modification times of the file pointed to by the *path* parameter to the value of the *times* parameter. It allows time specifications that are accurate to the nearest second.

The *times* parameter is a pointer to a **utimbuf** structure, which is defined in the **utime.h** header file. The **actime** field in this structure represents the date and time of last access, and the **modtime** field represents the date and time of last modification. The times in the **utimbuf** structure are measured in seconds since the Epoch, which is 00:00:00, January 1, 1970, Coordinated Universal Time (UTC).

If the *times* parameter is a null pointer, the access and modification times of the file are set to the current time. The effective user ID of the process either must be the same as the owner of the file, must have write access to the file, or must have appropriate privileges in order to use the call in this manner.

If the *times* parameter is not a null pointer, the access and modification times are set to the values contained in the designated structure. Only the owner of the file or a process with appropriate privileges can use the call this way.

Upon successful completion, the **utime( )** function marks the time of the last file status change, **st_ctime**, for update.

### Accessing Files in Restricted-Access Filesets

When accessing a file in a restricted-access fileset, the super ID (255,255 in the Guardian environment, 65535 in the OSS environment) is restricted by the same file permissions and owner privileges as any other user ID: It has no special privileges unless the executable file started by the super ID has the PRIVSETID file privilege. In this case, the process started by the super ID can switch to another ID and then access files in restricted-access filesets as that ID.

Executable files that have the PRIVSOARFOPEN privilege and that are started by a member of the Safeguard SECURITY-OSS-ADMINISTRATOR (SOA) group have the appropriate privilege to use this function on any file in a restricted-access fileset. However, Network File System (NFS) clients are not granted SOA group privileges, even if these clients are accessing the system with a user ID that is a member of the SOA security group.

For more information about restricted-access filesets and file privileges, see the *Open System Services Management and Operations Guide*.

### Use on Guardian Objects

The **utime( )** function is supported for Guardian files (that is, files within **/G**) that are unstructured Enscribe files. If the **utime( )** function is called for a Guardian file that has a small file label, the label is expanded to include the **st_atime** and **st_ctime** fields and to mark them for update.

The **utime( )** function cannot be used on a file in **/G** that is opened for execution. A call for such a file fails and **errno** is set to [ETXTBSY].

### Use From the Guardian Environment

The file access time is not updated by I/O operations that are performed on a file that was opened in the Guardian environment (that is, by the FILE_OPEN_ or OPEN Guardian procedures).

The **utime( )** function is one of a set of functions that have the following effects when the first of them is called from the Guardian environment:

- Two Guardian file system file numbers (not necessarily the next two available) are allocated for the root directory and the current working directory. These file numbers cannot be closed by calling the Guardian FILE_CLOSE_ procedure.
- The current working directory is assigned from the VOLUME attribute of the Guardian environment =_DEFAULTS DEFINE.
- The use of static memory by the process increases slightly.

These effects occur only when the first of the set of functions is called. The effects are not cumulative.

## NOTES

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

## RETURN VALUES

Upon successful completion, the value 0 (zero) is returned. Otherwise, the value -1 is returned, **errno** is set to indicate the error, and the file times are not changed.

## ERRORS

If any of the following conditions occurs, the **utime( )** function sets **errno** to the corresponding value:

[EACCES] One of the following conditions exists:

- Search permission is denied by a component of the pathname prefix.
- The *times* parameter is a null pointer, the effective user ID neither is the owner of the file nor has appropriate privileges, and write access is denied.

[EFAULT] Either the *path* parameter or the *times* parameter is an invalid address.

[EFSBAD] The fileset catalog is corrupted for the fileset involved in the requested operation.

[EINVAL] The function was called for a file in **/G** that is not a regular disk file.

[EIO] An input or output error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[ENAMETOOLONG]
One of the following is too long:

- The pathname pointed to by the *path* parameter
- A component of the pathname pointed to by the *path* parameter
- The intermediate result of pathname resolution when a symbolic link is part of the *path* parameter

The **pathconf( )** function can be called to obtain the applicable limits.

[ENOENT] One of the following conditions exists:

- The named file does not exist.
- The *path* parameter points to an empty string.
- The *path* parameter specifies a file on a remote HP NonStop node but communication with the remote node has been lost.

[ENOROOT] One of the following conditions exists:

- The root fileset of the local node (fileset 0) is not in the STARTED state.
- The current root fileset for the specified file is unavailable. The OSS name server for the fileset might have failed.
- The specified file is on a remote HP NonStop node and communication with the remote name server has been lost.

[ENOTDIR] A component of the pathname prefix is not a directory.

[ENOTSUP] The *path* parameter specifies a Guardian file on an SMF logical volume and one of the following conditions exists:

- The local system is running an RVU prior to J06.15 or H06.26.
- The *path* parameter specifies a file in /E and the remote system is running an RVU prior to J06.15 or H06.26.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[EOSSNOTRUNNING]
The OSS monitor process is not running.

[EPERM] One of the following conditions exist:

- The *times* parameter is not a null pointer, and the calling process has write access to the file but neither owns the file nor has appropriate privileges.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EROFS] The fileset that contains the file is mounted read-only.

[ETXTBSY] The *path* parameter specifies a file in the Guardian file system (**/G**) that is opened for execution.

**RELATED INFORMATION**

Functions: **stat(2)**.

**STANDARDS CONFORMANCE**

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EFAULT], [EFSBAD], [EINVAL], [ENOROOT], [ENOTSUP], [ENXIO], [EOSSNOTRUNNING], and [ETXTBSY] can be returned.

# Section 10.  System Functions (w)

This section contains reference pages for Open System Services (OSS) system function calls with names that begin with **w**. These reference pages reside in the **cat2** directory and are sorted alphabetically by U.S. English conventions in this section.

## NAME

**wait -** Waits for any child process to terminate

## LIBRARY

G-series native OSS processes: system library
H-series OSS processes: implicit libraries

## SYNOPSIS

**#include <sys/types.h>** /* optional except for POSIX.1 */
**#include <sys/wait.h>**

**pid_t wait(**
**int** **status_location***);**

## PARAMETERS

*status_location* Points to a location that receives the child process termination status, as defined in the **sys/wait.h** header file.

## DESCRIPTION

The **wait( )** function usually suspends the calling process until one of the following occurs:

- A child process initiates its own normal termination. That is, a child process calls the **_exit( )** or **exit( )** function or the Guardian STOP or PROCESS_STOP_ procedure on itself.
- A child process receives a signal that terminates the process. For example, some other process terminates the child process by calling the **kill( )** function or the Guardian STOP or PROCESS_STOP_ procedure against the child process.
- A child process terminates abnormally. The calling process receives a **SIGABEND** signal indicating that this process or another process has called the Guardian ABEND or PROCESS_STOP_ procedure specifying abnormal termination of the child process, or the child process has abnormally terminated for some other reason.
- The parent process catches a signal and invokes its own signal-catching function.

See the *Guardian Procedure Calls Reference Manual* for details on the Guardian ABEND, STOP, and PROCESS_STOP_ procedures.

The **wait( )** function returns without waiting if a child process that has not been waited for has already terminated prior to the call.

The effect of the **wait( )** function can be modified by the setting of the **SIGCHLD** signal. See the **sigaction(2)** reference page for details.

### Use With POSIX Threads

If Release Version Update (RVU) G06.21, or later, of T1248 POSIX threads is installed on the system, the T1248 version of **wait( )** is functionally equivalent to OSS **wait( )**, with the additional attribute of thread awareness. As such, it blocks only the thread calling it, without blocking any other threads. To call the T1248 **wait( )** function, include the linking flag **-l spt** when compiling thread-aware applications. If more than one thread is waiting on child processes, use the **spt_waitpid( )** function.

### Use From the Guardian Environment

If called from a Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

### Status Information

If the **wait( )** function returns because the status of a child process is available, it returns the OSS process ID of the child process. In this case, if the *status_location* parameter is not a null pointer, information is stored in the location pointed to by *status_location*.

The value stored at the location pointed to by *status_location* is 0 (zero) if and only if the status returned is from a terminated child process that either returned 0 (zero) from the **main( )** function or passed 0 (zero) as the *status* parameter to the **_exit( )** or **exit( )** function.

Regardless of its value, this status information can be interpreted using the following macros, which are defined in the **sys/wait.h** header file and evaluate to integer expressions:

**WCOMPLETION(**status_location*)**
Evaluates to the 16-bit Guardian completion code issued on process termination.

**WEXITSTATUS(**status_location*)**
If the value of **WIFEXITED(**status_location*)** is nonzero, evaluates to one of the following:

- The lower 8 bits of the *status* parameter that the child process passed to the **_exit( )** or **exit( )** function
- The lower 8 bits of the completion code for a process that terminated itself by calling the Guardian STOP or PROCESS_STOP_ procedure
- The lower 8 bits of the value that the child process returned from the **main( )** function

**WIFABENDED(**status_location*)**
Evaluates to a nonzero value if the child process terminated abnormally. A **SIGABEND** signal was received.

**WIFEXITED(**status_location*)**
Evaluates to a nonzero value if status was returned for a child process that terminated normally whether the termination was due to the **_exit( )** function, the **exit( )** function, the Guardian STOP procedure, or the Guardian PROCESS_STOP_ procedure.

**WIFSAVEABEND(**status_location*)**
Evaluates to a nonzero value if the terminated process created a saveabend file.

**WIFSIGNALED(**status_location*)**
Evaluates to a nonzero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught. Such a signal occurs, for example, when another process terminates the child process by calling the **kill( )** function, the Guardian STOP procedure, or the Guardian PROCESS_STOP_ procedure, or when the process abnormally terminates.

**WIFSTOPPED(**status_location*)**
Evaluates to a nonzero value if status was returned for a child process that is currently stopped.

This macro is normally only useful with the **waitpid( )** function.

**WSTOPSIG(*** *status_location* **)**
If the value of **WIFSTOPPED(*** *status_location* **)** is nonzero, evaluates to the number of the signal that caused the child process to stop.

This macro is normally only useful with the **waitpid( )** function.

**WTERMSIG(*** *status_location* **)**
If the value of **WIFSIGNALED(*** *status_location* **)** is nonzero, evaluates to the number of the signal that caused the termination of the child process.

See the *Guardian Procedure Calls Reference Manual* for details on the Guardian STOP and PROCESS_STOP_ procedures and on Guardian completion codes.

If the information stored at the location pointed to by the *status_location* parameter was stored there by a call to the **waitpid( )** function that specified the **WUNTRACED** option, exactly one of the **WIFEXITED**, **WIFSIGNALED**, and **WIFSTOPPED** macros evaluates to a nonzero value. If the information stored at the location pointed to by *status_location* was stored there by a call to the **wait( )** function, exactly one of the **WIFEXITED** and **WIFSIGNALED** macros evaluates to a nonzero value.

### Normal Self Termination

When a process terminates itself, information is returned to the parent process in the location pointed to by the *status_location* parameter. A process terminates itself in one of the following ways:

- Returning from its **main( )** function. The return value is placed in ******status_location*.
- Calling the **_exit( )** or **exit( )** function. The exit status is placed in ******status_location*.
- Calling the Guardian STOP or PROCESS_STOP_ procedure with parameters set for self-termination. The completion code is placed in ******status_location*.

The parent process can use the **WIFEXITED** macro to detect a child process that terminates itself; **WIFEXITED** evaluates to a nonzero value. The **WEXITSTATUS** macro evaluates to the lower 8 bits of the return value, exit status, or completion code. The **WCOMPLETION** macro evaluates to the full 16-bit completion code or to 16 bits of the 32-bit exit status code.

See the *Guardian Procedure Calls Reference Manual* for details on the Guardian STOP and PROCESS_STOP_ procedures and on Guardian completion codes.

### Termination by Another

The child process can be terminated by another process in one of the following ways:

- Another process calls the **kill( )** function with the OSS process ID of the child process.
- Another process calls the Guardian STOP procedure with the Guardian process ID of the child process or calls the Guardian PROCESS_STOP_ procedure with the Guardian process handle of the child process.

In either case, the **SIGKILL** signal is delivered. The parent process can use the **WIFSIGNALED** macro to detect when a signal causes the child process to terminate; **WIFSIGNALED** evaluates to a nonzero value. The **WTERMSIG** macro evaluates to the number of the signal that caused the termination. The **WCOMPLETION** macro evaluates to the completion code.

See the *Guardian Procedure Calls Reference Manual* for details on the Guardian STOP and PROCESS_STOP_ procedures and on Guardian completion codes.

### Abnormal Termination

Abnormal termination can occur for several reasons, including the following:

- The child process calls the Guardian ABEND procedure, or it calls the Guardian PROCESS_STOP_ procedure with the parameters set for abnormal termination.
- The processor in which the process was running fails.
- Some critical system resource is exhausted.
- One of the functions in the **exec**, **tdm_exec**, or **tdm_spawn** set of functions fails after the caller of that function has already been overlaid by the child process, and there is no caller to which it can return the error.
- Two traps occur inside an area where a Guardian trap handler is installed by the Guardian SETTRAP procedure.

In all cases of abnormal termination, the **SIGABEND** signal is delivered. Like the **SIGKILL** signal, **SIGABEND** can neither be caught nor ignored. Its default action is to terminate the process.

The **WIFSIGNALED** macro evaluates to a nonzero value and the **SIGABEND** signal is indicated by the **WTERMSIG** macro. Alternatively, the parent process can use the **WIFABENDED** macro to determine whether the child process terminated abnormally. The parent process can use the **WCOMPLETION** macro to read the completion code.

See the *Guardian Procedure Calls Reference Manual* for details on the Guardian ABEND, STOP, and PROCESS_STOP_ procedures and on Guardian completion codes.

### Saveabend File Creation

Whenever process termination is caused by signal delivery (that is, when the **WIFSIGNALED** macro evaluates to a nonzero value), it is possible that the terminating process creates a saveabend file.

A saveabend file is created for the process if the saveabend bit is set for the process in the process control block (PCB). This bit is set in any of the following ways:

- The compiler, linker, or Binder sets the saveabend bit in the code file header.
- The **tdm_fork( )**, **tdm_execve( )**, **tdm_execvep( )**, **tdm_spawn( )**, or **tdm_spawnp( )** function sets the **pe_debug_options** field of the **process_extensions_def** structure.
- The shell command that executes the process sets the saveabend bit.

If a saveabend file is created, the core dump (**CD**) bit is set in the information returned in the location pointed to by the *status_location* parameter. The parent process can use the **WIFSAVEABEND** macro to detect the creation of a saveabend file; **WIFSAVEABEND** evaluates to a nonzero value when the **CD** bit is set.

If a processor failure occurs, status about the terminated child processes in the failed processor is returned to the parent process in the location pointed to by the *status_location* parameter. In this case, no saveabend file is possible. **WIFSAVEABEND** evaluates to zero.

## NOTES

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes are assigned a parent process ID of 1.

Suspending a process is not always the same as stopping it. A process is only stopped when a job-control signal stops it.

## RETURN VALUES

If the **wait( )** function returns because the status of a child process is available, the OSS process ID of the child is returned to the calling process. If a signal is received via **pthread_kill(2)** that is not blocked,ignored, or handled, -1 is returned with an errno of EINTR.

Upon any error, the value -1 is returned and errno is set to indicate the error.

## ERRORS

If any of the following conditions occurs, the **wait( )** function sets **errno** to the corresponding value:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EFAULT] The buffer pointed to by the *status_location* parameter failed bounds checking.

[EINTR] The function was terminated by receipt of a signal. The information pointed to by the *status_location* parameter is not meaningful when this error occurs and should not be used in further processing.

[ENOTOSS] The calling process was not an OSS process. The **wait( )** function cannot be used in the Guardian environment.

## RELATED INFORMATION

Functions: **exec(2)**, **_exit(2)**, **exit(3)**, **fork(2)**, **spt_waitpid(2)**, **pause(3)**, **sigaction(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **waitpid(2)**.

## STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- The POSIX.1 standard states that when status information for two or more child processes is available, the order in which the information is returned by the **wait( )** function is unspecified. HP's implementation also does not provide this information in a specified sequence. The sequence should therefore not be depended upon for further processing.
- In addition to the status information mandated by the POSIX.1 standard, the HP implementation also returns status information for processes that terminate as a result of Guardian procedure calls. In addition, status is returned for processes that terminate abnormally as a result of a situation that is unique to NonStop server architecture, such as failure of the child process's processor while the parent process continues to execute.
- The POSIX.1 standard indicates that the value in the location pointed to by the *status_location* parameter is undefined when **errno** returns the value [EINTR]. HP's implementation also does not return meaningful information, and the value should not be used for further processing.

This function does not conform to the async-signal safe requirement of the POSIX.1 standard.

## NAME

**waitpid -** Waits for a specific child process to stop or terminate

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <sys/types.h>     /* optional except for POSIX.1 */
#include <sys/wait.h>

pid_t waitpid(
        pid_t process_id,
        int *status_location,
        int options);
```

## PARAMETERS

*process_id* Specifies the child process.

*status_location* Points to a location that receives the child process termination (or stop) status, as defined in the **sys/wait.h** header file.

*options* Modifies the behavior of the function.

## DESCRIPTION

The **waitpid( )** function usually suspends the calling process until one of the following occurs:

- The specified child process initiates its own normal termination. That is, the child process calls the **_exit( )** or **exit( )** function or the Guardian STOP or PROCESS_STOP_ procedure on itself.
- The child process receives a signal that terminates the process. For example, some other process terminates the child process by calling the **kill( )** function or the Guardian STOP or PROCESS_STOP_ procedure against the child process.
- The child process terminates abnormally. The calling process receives a **SIGABEND** signal indicating that this process or another process has called the Guardian ABEND or PROCESS_STOP_ procedure specifying abnormal termination of the child process, or the child process has abnormally terminated for some other reason.
- The child process was stopped (that is, suspended) by a job-control signal and the **WUNTRACED** option was set in this call to **waitpid( )**.
- The parent process catches a signal and invokes its own signal-catching function.

See the *Guardian Procedure Calls Reference Manual* for details on the Guardian ABEND, STOP, and PROCESS_STOP_ procedures.

The **waitpid( )** function returns without waiting if a child process that has not been waited for has already stopped or terminated prior to the call.

The POSIX.1 standard states that when status information for two or more child processes is available, the order in which the information is returned by the **waitpid( )** function is unspecified. HP’s implementation also does not provide this information in a reliable sequence. The sequence should therefore not be depended upon for further processing.

The effect of the **waitpid( )** function can be modified by the setting of the **SIGCHLD** signal. See the **sigaction(2)** reference page for details.

The **waitpid( )** function behaves identically to the **wait( )** function if the *process_id* parameter has the value -1 and the *options* parameter has the value 0 (zero). Otherwise, its behavior is modified by the values of the *process_id* and *options* parameters.

**Use From the Guardian Environment**

If called from a Guardian process, the actions of this function are undefined and **errno** is set to [ENOTOSS].

**Specifying the Child Process**

The **waitpid( )** function allows the calling process to gather status from a specific set of child processes. The **waitpid( )** function returns the status of a child process from this set. The *process_id* parameter specifies the set according to the following rules:

- If *process_id* is equal to -1, status is requested for any child process. In this respect, the **waitpid( )** function is equivalent to the **wait( )** function.
- If *process_id* is greater than 0 (zero), it specifies the OSS process ID (PID) of a single child process for which status is requested.
- If *process_id* is equal to 0 (zero), status is requested for any child process whose process group ID is equal to that of the calling process.
- If *process_id* is less than -1, status is requested for any child process whose process group ID is equal to the absolute value of *process_id*.

**Options**

The *options* parameter modifies the behavior of the **waitpid( )** function. This parameter is constructed from the bitwise-inclusive OR of the following flag values:

**WNOHANG** Prevents the calling process from being suspended even if there are child processes to wait for. In this case, 0 (zero) is returned, indicating that there are no child processes that have stopped or terminated.

**WUNTRACED**

Returns information when child processes of the current process are stopped because they received a **SIGTTIN**, **SIGTTOU**, **SIGSTOP**, or **SIGTSTP** signal.

**Status Information**

If the **waitpid( )** function returns because the status of a child process is available, it returns the OSS process ID of the child process. In this case, if the *status_location* parameter is not a null pointer, information is stored in the location pointed to by *status_location*.

The value stored at the location pointed to by *status_location* is 0 (zero) if and only if the status returned is from a terminated child process that either returned 0 (zero) from the **main( )** function or passed 0 (zero) as the *status* parameter to the **_exit( )** or **exit( )** function.

Regardless of its value, this status information can be interpreted using the following macros, which are defined in the **sys/wait.h** header file and evaluate to integer expressions:

**WCOMPLETION(∗*status_location*)**

Evaluates to the 16-bit Guardian completion code issued on process termination.

**WEXITSTATUS(∗*status_location*)**
If the value of **WIFEXITED(∗*status_location*)** is nonzero, evaluates to one of the following:

- The lower 8 bits of the *status* parameter that the child process passed to the **_exit( )** or **exit( )** function
- The lower 8 bits of the completion code for a process that terminated itself by calling the Guardian STOP or PROCESS_STOP_ procedure
- The lower 8 bits of the value that the child process returned from the **main( )** function

**WIFABENDED(∗*status_location*)**
Evaluates to a nonzero value if the child process terminated abnormally. A **SIGABEND** signal was received.

**WIFEXITED(∗*status_location*)**
Evaluates to a nonzero value if status was returned for a child process that terminated normally whether the termination was due to the **_exit( )** function, the **exit( )** function, the Guardian STOP procedure, or the Guardian PROCESS_STOP_ procedure.

**WIFSAVEABEND(∗*status_location*)**
Evaluates to a nonzero value if the terminated process created a saveabend file.

**WIFSIGNALED(∗*status_location*)**
Evaluates to a nonzero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught. Such a signal occurs, for example, when another process terminates the child process by calling the **kill( )** function, the Guardian STOP procedure, or the Guardian PROCESS_STOP_ procedure, or when the process abnormally terminates.

**WIFSTOPPED(∗*status_location*)**
Evaluates to a nonzero value if status was returned for a child process that is currently stopped.

This macro returns a nonzero value only when the **WUNTRACED** option was set in the call to **waitpid( )** and the stopped process was not previously reported.

**WSTOPSIG(∗*status_location*)**
If the value of **WIFSTOPPED(∗*status_location*)** is nonzero, evaluates to the number of the signal that caused the child process to stop.

**WTERMSIG(∗*status_location*)**
If the value of **WIFSIGNALED(∗*status_location*)** is nonzero, evaluates to the number of the signal that caused the termination of the child process.

See the *Guardian Procedure Calls Reference Manual* for details on the Guardian STOP and PROCESS_STOP_ procedures and on Guardian completion codes.

If the information stored at the location pointed to by the *status_location* parameter was stored there by a call to the **waitpid( )** function that specified the **WUNTRACED** option, exactly one of the **WIFEXITED**, **WIFSIGNALED**, and **WIFSTOPPED** macros evaluates to a nonzero value. If the information stored at the location pointed to by *status_location* was stored there by a call to **waitpid( )** that did not specify the **WUNTRACED** option or by a call to the **wait( )** function, exactly one of the **WIFEXITED** and **WIFSIGNALED** macros evaluates to a nonzero value.

**Normal Self Termination**

When a process terminates itself, information is returned to the parent process in the location pointed to by the *status_location* parameter. A process terminates itself in one of the following ways:

- Returning from its **main( )** function. The return value is placed in *∗status_location*.
- Calling the **_exit( )** or **exit( )** function. The exit status is placed in *∗status_location*.
- Calling the Guardian STOP or PROCESS_STOP_ procedure with parameters set for self-termination. The completion code is placed in *∗status_location*.

The parent process can use the **WIFEXITED** macro to detect a child process that terminates itself; **WIFEXITED** evaluates to a nonzero value. The **WEXITSTATUS** macro evaluates to the lower 8 bits of the return value, exit status, or completion code. The **WCOMPLETION** macro evaluates to the full 16-bit completion code or to 16 bits of the 32-bit exit status code.

See the *Guardian Procedure Calls Reference Manual* for details on the Guardian STOP and PROCESS_STOP_ procedures and on Guardian completion codes.

**Termination by Another**

The child process can be terminated by another process in one of the following ways:

- Another process calls the **kill( )** function with the OSS process ID of the child process.
- Another process calls the Guardian STOP procedure with the Guardian process ID of the child process or calls the Guardian PROCESS_STOP_ procedure with the Guardian process handle of the child process.

In either case, the **SIGKILL** signal is delivered. The parent process can use the **WIFSIGNALED** macro to detect when a signal causes the child process to terminate; **WIFSIGNALED** evaluates to a nonzero value. The **WTERMSIG** macro evaluates to the number of the signal that caused the termination. The **WCOMPLETION** macro evaluates to the completion code.

See the *Guardian Procedure Calls Reference Manual* for details on the Guardian STOP and PROCESS_STOP_ procedures and on Guardian completion codes.

**Abnormal Termination**

Abnormal termination can occur for several reasons, including the following:

- The child process calls the Guardian ABEND procedure, or it calls the Guardian PROCESS_STOP_ procedure with the parameters set for abnormal termination.
- The processor in which the process was running fails.
- Some critical system resource is exhausted.
- One of the functions in the **exec**, **tdm_exec**, or **tdm_spawn** set of functions fails after the caller of that function has already been overlaid by the child process, and there is no caller to which it can return the error.
- Two traps occur inside an area where a Guardian trap handler is installed by the Guardian SETTRAP procedure.

In all cases of abnormal termination, the **SIGABEND** signal is delivered. Like the **SIGKILL** signal, **SIGABEND** can neither be caught nor ignored. Its default action is to terminate the process.

The **WIFSIGNALED** macro evaluates to a nonzero value and the **SIGABEND** signal is indicated by the **WTERMSIG** macro. Alternatively, the parent process can use the **WIFABENDED** macro to determine whether the child process terminated abnormally. The parent process can use the **WCOMPLETION** macro to read the completion code.

See the *Guardian Procedure Calls Reference Manual* for details on the Guardian ABEND, STOP, and PROCESS_STOP_ procedures and on Guardian completion codes.

**Process Stopped**

If the **WUNTRACED** option is set in the **waitpid( )** call, the call returns when the child process is temporarily suspended because it received a **SIGTTIN**, **SIGTTOU**, **SIGSTOP**, or **SIGTSTOP** signal.

The **WIFSTOPPED** macro evaluates to a nonzero value. The **WSTOPSIG** macro evaluates to the number of the signal that caused the process to stop.

**Saveabend File Creation**

Whenever process termination is caused by signal delivery (that is, when the **WIFSIGNALED** macro evaluates to a nonzero value), it is possible that the terminating process creates a saveabend file.

A saveabend file is created for the process if the saveabend bit is set for the process in the process control block (PCB). This bit is set in any of the following ways:

- The compiler, linker, or Binder sets the saveabend bit in the code file header.
- The **tdm_fork( )**, **tdm_execve( )**, **tdm_execvep( )**, **tdm_spawn( )**, or **tdm_spawnp( )** function sets the **pe_debug_options** field of the **process_extensions_def** structure.
- The shell command that executes the process sets the saveabend bit.

If a saveabend file is created, the core dump (**CD**) bit is set in the information returned in the location pointed to by the *status_location* parameter. The parent process can use the **WIFSAVEABEND** macro to detect the creation of a saveabend file; **WIFSAVEABEND** evaluates to a nonzero value when the **CD** bit is set.

If a processor failure occurs, status about the terminated child processes in the failed processor is returned to the parent process in the location pointed to by the *status_location* parameter. In this case, no saveabend file is possible. **WIFSAVEABEND** evaluates to zero.

## NOTES

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes are assigned a parent process ID of 1.

Suspending a process is not always the same as stopping it. A process is only stopped when a job-control signal stops it.

To use the **waitpid( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_waitpid(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit threaded applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

### RETURN VALUES

If the **waitpid( )** function returns because the status of a child process is available, the OSS process ID of the child is returned to the calling process. If the function returns because a signal was caught by the calling process, the value -1 is returned and **errno** is set to [EINTR]. Upon any error, the value -1 is returned and **errno** is set to indicate the error.

If the **WNOHANG** value of the *options* parameter was specified and there are no stopped or exited child processes, the **waitpid( )** function returns the value 0 (zero).

### ERRORS

If any of the following conditions occurs, the **waitpid( )** function sets **errno** to the corresponding value:

[ECHILD] The process or process group ID specified by the *process_id* parameter either does not exist or is not a child process of the calling process.

[EFAULT] The buffer pointed to by the *status_location* parameter failed bounds checking.

[EINTR] The function was terminated by receipt of a signal. The information pointed to by the *status_location* parameter is not meaningful when this error occurs and should not be used in further processing.

This error is also returned if the **waitpid( )** function is thread-aware and a signal received from the **pthread_kill( )** function is not blocked, ignored, or handled.

[EINVAL] The value of the *options* parameter is invalid.

[ENOTOSS] The calling process was not an OSS process. The **waitpid( )** function cannot be used in the Guardian environment.

### RELATED INFORMATION

Functions: **exec(2)**, **_exit(2)**, **exit(3)**, **fork(2)**, **pause(3)**, **sigaction(2)**, **spt_waitpid(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**, **wait(2)**.

### STANDARDS CONFORMANCE

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- The POSIX.1 standard states that when status information for two or more child processes is available, the order in which the information is returned by the **waitpid( )** function is unspecified. HP's implementation also does not provide this information in a specified sequence. The sequence should therefore not be depended upon for further processing.

- In addition to the status information mandated by the POSIX.1 standard, the HP implementation also returns status information for processes that terminate as a result of Guardian procedure calls. In addition, status is returned for processes that terminate abnormally as a result of a situation that is unique to HP NonStop server architecture, such as failure of the child process's processor while the parent process continues to execute.
- The POSIX.1 standard indicates that the value in the location pointed to by the *status_location* parameter is undefined when **errno** returns the value [EINTR]. HP's implementation also does not return meaningful information, and the value should not be used for further processing.

This function is an extension to the XPG4 Version 2 specification.

The use of this function with the POSIX User Thread Model library conforms to the following industry standards:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface

This function does not conform to the async-signal safe requirement of the POSIX.1 standard.

## NAME

**write -** Writes to a file

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <sys/types.h>      /* optional except for POSIX.1 */
#include <unistd.h>

ssize_t write(
        int filedes,
        void *buffer,
        size_t nbytes);
```

## PARAMETERS

*filedes*
: Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

  When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function, or the thread-aware **accept( )**, **dup2( )**, or **fcntl( )** function.

*buffer*
: Identifies the buffer containing the data to be written.

*nbytes*
: Specifies the number of bytes to write.

## DESCRIPTION

The **write( )** function attempts to write *nbytes* of data to the file associated with the *filedes* parameter from the buffer pointed to by the *buffer* parameter.

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **write )** or **write64_( )** may be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **write64_( )** must be called.

32-bit Guardian and 64-bit OSS clients can pass 32-bit pointers and 64-bit pointers to **write64_( )**.

For all regular and non-regular files, if the value of the *nbytes* parameter is 0 (zero) and the value of *filedes* is a valid file descriptor, the **write( )** function returns 0 (zero).

The appropriate file time fields are updated unless *nbytes* is 0 (zero).

With regular files and devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. If this incremented file pointer is greater than the length of the file, the length of the file is set to this file offset. Upon return from the **write( )** function, the file pointer is incremented by the number of bytes actually written.

With devices incapable of seeking, writing always takes place starting at the current position. For such devices, the value of the file pointer after a call to the **write( )** function is always 0 (zero).

Fewer bytes than requested can be written if there is not enough room to satisfy the request. In this case, the number of bytes written is returned. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write request of 512 bytes returns a value of 20. The limit reached can be either the end of the physical medium or the value that has been set by the **ulimit( )** function. The next write of a nonzero number of bytes gives a failure return (except as noted later).

Upon successful completion, the **write( )** function returns the number of bytes actually written to the file associated with *filedes*. This number is never greater than the value of *nbytes*.

If the **O_APPEND** flag of the file status is set, the file offset is set to the end of the file prior to each write.

Write requests to a pipe or a FIFO file are handled the same as writes to a regular file with these exceptions:

- No file offset is associated with a pipe; therfore, each **write( )** request appends to the end of the pipe.
- If the size of the **write( )** request is less than or equal to the value of the **PIPE_BUF** system variable, the **write( )** function is guaranteed to be atomic. The data is not interleaved with data from other processes doing writes on the same pipe.
- If the size of the **write( )** request is greater than the value of the **PIPE_BUF** system variable, the file system attempts to resize the pipe buffer from 2 * **PIPE_BUF** to 65,536 bytes. If the resizing is successful, the file system performs atomic writes of up to 32,768 bytes and can transfer up to 52 kilobytes of data from the pipe buffer on subsequent **read( )** or **readv( )** calls by the client.

  If the file system cannot resize the buffer, it continues to use the existing buffer. A second attempt at resizing occurs after approximately a minute elapses.

  Writes of greater than **PIPE_BUF** bytes can have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the **O_NONBLOCK** flag is set.
- If the **O_NONBLOCK** flag is not set, a **write( )** request to a full pipe causes the process to block until enough space becomes available to handle the entire request.
- If the **O_NONBLOCK** flag is set, **write( )** requests are handled differently in these ways:
  - The **write( )** function does block the process.
  - **write( )** requests for **PIPE_BUF** or fewer bytes either succeed completely and return the value of the *nbytes* parameter, or return the value -1 and set **errno** to [EAGAIN].
  - A **write( )** request for greater than **PIPE_BUF** bytes either transfers what it can and returns the number of bytes written, or transfers no data and returns the value -1 with **errno** set to [EAGAIN]. Also, if a request is greater than **PIPE_BUF** bytes and all data previously written to the pipe has been read, **write( )** transfers at least **PIPE_BUF** bytes.

When attempting to write to a file descriptor for a special character device (a terminal) that cannot accept data immediately:

- If the **O_NONBLOCK** flag is clear, the **write( )** function blocks until the data can be accepted or an error occurs.
- If the **O_NONBLOCK** flag is set, the **write( )** function returns the value -1 and **errno** is set to [EAGAIN].

When attempting to write to a socket and with no space available for data:

- If the **O_NONBLOCK** flag is not set, the **write( )** function blocks until space becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **write( )** function returns the value -1 and sets **errno** to [EWOULDBLOCK].

Upon successful completion, the **write( )** function marks the **st_ctime** and **st_mtime** fields of the file for update and clears the set-user-ID and set-group-ID attributes if the file is a regular file.

The **fcntl( )** function provides more information about record locks.

If the **write( )** function is interrupted by a signal before it writes any data, it returns the value -1 with **errno** set to [EINTR]. If the **write( )** function is interrupted by a signal after it has successfully written some data, it returns the number of bytes that it has written.

**Use on Guardian Objects**

Attempting to write to a Guardian file (that is, a file in **/G**) that is locked causes the **write( )** function to return -1 and set **errno** to [EGUARDIANLOCKED].

**Use From a Threaded Application**

The thread-aware **write( )** function behaves exactly the same as **spt_writez( )** in the Standard POSIX Threads library. For file descriptors for regular files, if this thread-aware **write( )** function must wait for an I/O operation to complete on an open file, this function blocks the thread (instead of the entire process) that called it, while it waits for the I/O operation to complete.

## NOTES

To use the **write( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_writez(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running J06.10 or later RVUs or H06.21 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to enable the function on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application that uses the POSIX User Thread Model library on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the **write( )** function returns the number of bytes that were actually written. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **write( )** function sets **errno** to the corresponding value:

[EAGAIN] One of these conditions exists:

- An attempt was made to write to a file descriptor that cannot accept data, and the **O_NONBLOCK** flag is set.
- A write to a pipe (FIFO file) of **PIPE_BUF** bytes or less is requested, **O_NONBLOCK** is set, and fewer than *nbytes* of free space are available.
- The **O_NONBLOCK** flag is set on this file, and the process would be delayed in the write operation.

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function is in progress on a regular file and a function that is process-blocking for regular files attempts to begin an I/O operation on the same open file.

If the **write( )** function is thread-aware, the [EALREADY] value is not returned.

[EBADF] The *filedes* parameter does not specify a valid file descriptor open for writing.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The *buffer* parameter points to a location outside of the allocated address space of the process.

[EFBIG] The application is attempting to write at or beyond the file offset maximum established when the file was opened.

[EGUARDIANLOCKED]
A **write( )** operation was attempted to a file in the Guardian file system (that is, a file in **/G**) that is locked.

[EINTR] A **write( )** operation was interrupted by a signal before any data was written.

[EINVAL] One of these conditions occurred:

- The file position pointer associated with the file specified by the *filedes* parameter was negative.
- The value of the *nbytes* parameter is greater than **SSIZE_MAX**.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
- A physical I/O error occurred. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOSPC] No free space is left on the fileset containing the file.

[ENOTCONN] An attempt was made to write to a socket that is not connected to a peer socket.

[ENXIO] One of these conditions occurred:

- The device associated with the file descriptor specified by the *filedes* parameter is a block special device or character special file, and the file pointer is out of range.
- No existing device is associated with the file descriptor specified by the *filedes* parameter.

[EPIPE] One of these conditions occurred:

- An attempt was made to write to a pipe or FIFO file that is not open for reading by any process. A **SIGPIPE** signal is sent if the process is running in the OSS environment.
- An attempt was made to write to a pipe that has only one end open.
- An attempt was made to write to a socket that is shut down or closed.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWOULDBLOCK]
The process attempted an operation on a socket for which **O_NONBLOCK** is set, there is no space available, and no error has occurred.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.

- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **creat(2)**, **creat64(2)**, **fcntl(2)**, **lseek(2)**, **lseek64(2)**, **open(2)**, **open64(2)**, **pipe(2)**, **socket(2)**, **spt_writez(2)**, **ulimit(3)**.

## STANDARDS CONFORMANCE

The HP implementation does not generate the **SIGXFSZ** signal.

The POSIX standards leave some features to the implementing vendor to define. These features are affected in the HP implementation:

- Calls to the **write( )** function with the *nbytes* parameter equal to 0 are supported for all regular and nonregular files.
- After reading from a device that is incapable of seeking, the value of the file pointer is always 0 (zero).
- Specifying a value for the *nbytes* parameter that is greater than **SSIZE_MAX** causes the **write( )** function to return -1 and set **errno** to [EINVAL].
- **errno** can be set to [EIO] if a physical I/O error occurs.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [ECONNRESET], [EFAULT], [EGUARDIANLOCKED], [EINVAL], [ENETDOWN], [ENOTCONN], [ETIMEDOUT], and [EWRONGID] can be returned.
- For systems running J06.07 and later J-series RVUs or H06.18 and later H-series RVUs, the **errno** value [ENOMEM] can be returned when there is not enough system memory available to complete the operation.

The use of this function with the POSIX User Thread Model library conforms to industry standards as follows:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface
- When a signal arrives during a call to a thread-aware **write( )** function, the thread-aware **write( )** retries the I/O operation instead of returning the **errno** value [EINTR] with the following exception. If the thread-aware **fork( )** function is called by a signal handler that is running on a thread performing a thread-aware **write( )** call, the thread-aware **write( )** call in the child process returns [EINTR] to the application.

## NAME

**write64_** - Writes to a file

## LIBRARY

G-series native OSS processes: system library
H-series and J-series OSS processes: implicit libraries
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <sys/types.h>    /* optional except for POSIX.1 */
#include <unistd.h>

long long write64_(
        int filedes,
        void _ptr64 *buffer,
        unsigned long long nbytes);
```

## PARAMETERS

*filedes*
Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

*buffer*
Identifies the buffer containing the data to be written.

*nbytes*
Specifies the number of bytes to write.

## DESCRIPTION

The **write64_( )** function attempts to write *nbytes* of data to the file associated with the *filedes* parameter from the buffer pointed to by the *buffer* parameter.

To pass a 32-bit pointer from a 32-bit Guardian or OSS client, **write( )** or **write64_( )** may be called.

To pass a 64-bit pointer from a 32-bit Guardian or OSS client, **write64_( )** must be called.

32-bit Guardian and 64-bit OSS clients can pass 32-bit pointers and 64-bit pointers to **write64_( )**.

For all regular and non-regular files, if the value of the *nbytes* parameter is 0 (zero) and the value of *filedes* is a valid file descriptor, the **write64_( )** function returns 0 (zero).

The appropriate file time fields are updated unless *nbytes* is 0 (zero).

With regular files and devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. If this incremented file pointer is greater than the length of the file, the length of the file is set to this file offset. Upon return from the **write64_( )** function, the file pointer is incremented by the number of bytes actually written.

With devices incapable of seeking, writing always takes place starting at the current position. For such devices, the value of the file pointer after a call to the **write64_( )** function is always 0 (zero).

Fewer bytes than requested can be written if there is not enough room to satisfy the request. In this case, the number of bytes written is returned. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write request of 512 bytes returns a value of 20. The limit reached can be either the end of the physical medium or the value that has been set by the **ulimit( )** function. The next write of a nonzero number of bytes gives a failure return (except as noted later).

Upon successful completion, the **write64_( )** function returns the number of bytes actually written to the file associated with *filedes*. This number is never greater than the value of *nbytes*.

If the **O_APPEND** flag of the file status is set, the file offset is set to the end of the file prior to each write.

Write requests to a pipe or a FIFO file are handled the same as writes to a regular file with these exceptions:

- No file offset is associated with a pipe; therfore, each **write64_( )** request appends to the end of the pipe.
- If the size of the **write64_( )** request is less than or equal to the value of the **PIPE_BUF** system variable, the **write64_( )** function is guaranteed to be atomic. The data is not interleaved with data from other processes doing writes on the same pipe.
- If the size of the **write64_( )** request is greater than the value of the **PIPE_BUF** system variable, the file system attempts to resize the pipe buffer from 2 * **PIPE_BUF** to 65,536 bytes. If the resizing is successful, the file system performs atomic writes of up to 32,768 bytes and can transfer up to 52 kilobytes of data from the pipe buffer on subsequent **read( )**, **read64_( )**, or **readv( )** calls by the client.

  If the file system cannot resize the buffer, it continues to use the existing buffer. A second attempt at resizing occurs after approximately a minute elapses.

  Writes of greater than **PIPE_BUF** bytes can have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the **O_NONBLOCK** flag is set.
- If the **O_NONBLOCK** flag is not set, a **write64_( )** request to a full pipe causes the process to block until enough space becomes available to handle the entire request.
- If the **O_NONBLOCK** flag is set, **write64_( )** requests are handled differently in these ways:
  - The **write64_( )** function does block the process.
  - **write64_( )** requests for **PIPE_BUF** or fewer bytes either succeed completely and return the value of the *nbytes* parameter, or return the value -1 and set **errno** to [EAGAIN].
  - A **write64_( )** request for greater than **PIPE_BUF** bytes either transfers what it can and returns the number of bytes written, or transfers no data and returns the value -1 with **errno** set to [EAGAIN]. Also, if a request is greater than **PIPE_BUF** bytes and all data previously written to the pipe has been read, **write64_( )** transfers at least **PIPE_BUF** bytes.

When attempting to write to a file descriptor for a special character device (a terminal) that cannot accept data immediately:

- If the **O_NONBLOCK** flag is clear, the **write64_( )** function blocks until the data can be accepted or an error occurs.
- If the **O_NONBLOCK** flag is set, the **write64_( )** function returns the value -1 and **errno** is set to [EAGAIN].

When attempting to write to a socket and with no space available for data:

- If the **O_NONBLOCK** flag is not set, the **write64_( )** function blocks until space becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **write64_( )** function returns the value -1 and sets **errno** to [EWOULDBLOCK].

Upon successful completion, the **write64_( )** function marks the **st_ctime** and **st_mtime** fields of the file for update and clears the set-user-ID and set-group-ID attributes if the file is a regular file.

The **fcntl( )** function provides more information about record locks.

If the **write64_( )** function is interrupted by a signal before it writes any data, it returns the value -1 with **errno** set to [EINTR]. If the **write64_( )** function is interrupted by a signal after it has successfully written some data, it returns the number of bytes that it has written.

### Use on Guardian Objects

Attempting to write to a Guardian file (that is, a file in **/G**) that is locked causes the **write64_( )** function to return -1 and set **errno** to [EGUARDIANLOCKED].

## NOTES

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

## RETURN VALUES

Upon successful completion, the **write64_( )** function returns the number of bytes that were actually written. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

## ERRORS

If any of these conditions occurs, the **write64_( )** function sets **errno** to the corresponding value:

[EAGAIN] One of these conditions exists:

- An attempt was made to write to a file descriptor that cannot accept data, and the **O_NONBLOCK** flag is set.
- A write to a pipe (FIFO file) of **PIPE_BUF** bytes or less is requested, **O_NONBLOCK** is set, and fewer than *nbytes* of free space are available.
- The **O_NONBLOCK** flag is set on this file, and the process would be delayed in the write operation.

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function (such as **spt_writez( )**)is in progress on a regular file and a function that is process-blocking for regular files (such as **read( )**, **spt_read( )**, or **spt_readx( )**) attempts to begin an I/O operation on the same open file.

[EBADF] The *filedes* parameter does not specify a valid file descriptor open for writing.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] The *buffer* parameter points to a location outside of the allocated address space of the process.

[EFBIG] The application is attempting to write at or beyond the file offset maximum established when the file was opened.

[EGUARDIANLOCKED]
A **write64_( )** operation was attempted to a file in the Guardian file system (that is, a file in **/G**) that is locked.

[EINTR] A **write64_( )** operation was interrupted by a signal before any data was written.

[EINVAL] One of these conditions occurred:

- The file position pointer associated with the file specified by the *filedes* parameter was negative.
- The value of the *nbytes* parameter is greater than **SSIZE_MAX**.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
- A physical I/O error occurred. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOSPC] No free space is left on the fileset containing the file.

[ENOTCONN] An attempt was made to write to a socket that is not connected to a peer socket.

[ENXIO] One of these conditions occurred:

- The device associated with the file descriptor specified by the *filedes* parameter is a block special device or character special file, and the file pointer is out of range.
- No existing device is associated with the file descriptor specified by the *filedes* parameter.

[EPIPE] One of these conditions occurred:

- An attempt was made to write to a pipe or FIFO file that is not open for reading by any process. A **SIGPIPE** signal is sent if the process is running in the OSS environment.

- An attempt was made to write to a pipe that has only one end open.
- An attempt was made to write to a socket that is shut down or closed.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWOULDBLOCK]
The process attempted an operation on a socket for which **O_NONBLOCK** is set, there is no space available, and no error has occurred.

[EWRONGID] One of these conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

**RELATED INFORMATION**

Functions: **creat(2)**, **creat64(2)**, **fcntl(2)**, **lseek(2)**, **lseek64(2)**, **open(2)**, **open64(2)**, **pipe(2)**, **read(2)**, **read64_(2)**, **socket(2)**, **ulimit(3)**, **write(2)**.

**STANDARDS CONFORMANCE**

This API is an HP extension and is not standards conformant.

## NAME

**writev -** Writes to a file from scattered buffers

## LIBRARY

G-series native OSS processes: **/G/system/sys***nn***/zossesrl**
32-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/zossedll**
64-bit H-series and J-series OSS processes: **/G/system/zdll***nnn***/yossedll**
32-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/zputdll**
64-bit H-series and J-series OSS processes that use the POSIX User Thread Model library:
**/G/system/zdll***nnn***/yputdll**

## SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>

int writev(
        int filedes,
        struct iovec *iov,
        int iov_count);
```

## PARAMETERS

*filedes*
: Specifies an open file descriptor obtained from a successful call to the **accept( )**, **creat( )**, **creat64( )**, **dup( )**, **dup2( )**, **fcntl( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function.

  When the function is thread-aware, specifies an open file descriptor obtained from a successful call to the **creat( )**, **creat64( )**, **dup( )**, **open( )**, **open64( )**, **pipe( )**, **socket( )**, or **socketpair( )** function, or the thread-aware **accept( )**, **dup2( )**, or **fcntl( )** function.

*iov*
: Points to a **iovec** structure that identifies the buffers containing the data to be written.

*iov_count*
: Specifies the number of **iovec** structure entries (buffers) pointed to by the *iov* parameter.

## DESCRIPTION

The **writev( )** function attempts to write data to the file associated with the *filedes* parameter from the set of buffers pointed to by the *iov* parameter.

The **writev( )** function performs the same action as the **write( )** function, but gathers the output data from the *iov_count* buffers specified by the **iovec** structure buffers pointed to by the *iov* parameter.

The **iovec** structure is defined in the **sys/uoi.h** header file and contains entries with these members:

```
caddr_t  iov_base;
int      iov_len;
```

The **iov_base** and **iov_len** members of each **iovec** structure entry specify the base address and length of an area in memory from which data should be written. The **writev( )** function always writes a complete buffer before proceeding to the next.

With regular files and devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. If this incremented file pointer is greater than the length of the file, the length of the file is set to this file offset. Upon return from the **writev( )** function, the file pointer is incremented by the number of bytes actually written.

With devices incapable of seeking, writing always takes place starting at the current position. For such devices, the value of the file pointer after a call to the **writev( )** function is always 0 (zero).

Fewer bytes than requested can be written if there is not enough room to satisfy the request. In this case, the number of bytes written is returned. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write request of 512 bytes returns a value of 20. The limit reached can be either the end of the physical medium or the value that has been set by the **ulimit( )** function. The next write of a nonzero number of bytes gives a failure return (except as noted later).

Upon successful completion, the **writev( )** function returns the number of bytes actually written to the file associated with *filedes*.

If the **O_APPEND** status flag of the file is set, the file offset is set to the end of the file prior to each write.

Write requests to a pipe or FIFO file are handled the same as writes to a regular file with these exceptions:

- No file offset is associated with a pipe; therfore, each **writev( )** request appends to the end of the pipe.
- If the size of the **writev( )** request is less than or equal to the value of the **PIPE_BUF** system variable, the **writev( )** function is guaranteed to be atomic. The data is not interleaved with data from other processes doing writes on the same pipe.
- If the size of the **writev( )** request is greater than the value of the **PIPE_BUF** system variable, the file system attempts to resize the pipe buffer from 2 * **PIPE_BUF** to 65,536 bytes. If the resizing is successful, the file system performs atomic writes of up to 32,768 bytes and can transfer up to 52 kilobytes of data from the pipe buffer on subsequent **read( )** or **readv( )** calls by the client.

  If the file system cannot resize the buffer, it continues to use the existing buffer. A second attempt at resizing occurs after approximately a minute elapses.

  Writes of greater than **PIPE_BUF** bytes can have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the **O_NONBLOCK** flag is set.
- If the **O_NONBLOCK** flag is not set, a **writev( )** request to a full pipe causes the process to block until enough space becomes available to handle the entire request.
- If the **O_NONBLOCK** flag is set, **writev( )** requests are handled differently in these ways:
  - The **writev( )** function does block the process.
  - The **writev( )** requests for **PIPE_BUF** or fewer bytes either succeed completely and return the number of bytes written, or return the value -1 and set **errno** to [EAGAIN].
  - A **writev( )** request for greater than **PIPE_BUF** bytes either transfers what it can and returns the number of bytes written, or transfers no data and returns the value -1 with **errno** set to [EAGAIN]. Also, if a request is greater than **PIPE_BUF** bytes and all data previously written to the pipe has been read, **writev( )** transfers at least **PIPE_BUF** bytes.

When attempting to write to a file descriptor for a special character device (a terminal) that cannot accept data immediately:

- If the **O_NONBLOCK** flag is clear, the **writev( )** function blocks until the data can be accepted or an error occurs.
- If the **O_NONBLOCK** flag is set, the **writev( )** function returns the value -1 and **errno** is set to [EAGAIN].

When attempting to write to a socket with no space available for data:

- If the **O_NONBLOCK** flag is not set, the **writev( )** function blocks until space becomes available or an error occurs.
- If the **O_NONBLOCK** flag is set, the **writev( )** function returns the value -1 and sets **errno** to [EWOULDBLOCK].

Upon successful completion, the **writev( )** function marks the **st_ctime** and **st_mtime** fields of the file for update and clears the set-user-ID and set-group-ID attributes if the file is a regular file.

The **fcntl( )** function provides more information about record locks.

If it is interrupted by a signal before it writes any data, the **writev( )** function returns the value -1 with **errno** set to [EINTR]. If it is interrupted by a signal after it has successfully written some data, the **writev( )** function returns the number of bytes that it has written.

### Use on Guardian Objects

Attempting to write to a Guardian file (that is, a file in **/G**) that is locked causes the **writev( )** function to return -1 and set **errno** to [EGUARDIANLOCKED].

### Use From a Threaded Application

The thread-aware **writev( )** function behaves exactly the same as **spt_writevz( )** in the Standard POSIX Threads library. For file descriptors for regular files, if this thread-aware **writev( )** function must wait for an I/O operation to complete on an open file, this function blocks the thread (instead of the entire process) that called it, while it waits for the I/O operation to complete.

## NOTES

To use the **writev( )** functionality in a threaded application that uses the Standard POSIX Threads library, see **spt_writevz(2)**.

To use this function in a threaded application that uses the POSIX User Thread Model library on systems running H06.21 or later RVUs or J06.10 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.
- Link the application to the **zputdll** library (**/G/system/zdll***nnn***/zputdll**).

On systems running H06.24 or later H-series RVUs or J06.13 or later J-series RVUs, you can use this function with 32-bit or 64-bit OSS applications.

To use this function in a 32-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, perform the same tasks (described above) used to make the function thread-aware in a multi-threaded application on systems running H06.21/J06.10 or later RVUs.

To use this function in a 64-bit threaded application on systems running H06.24 or later RVUs or J06.13 or later RVUs, you must perform all of the following tasks to make the function thread-aware in a multi-threaded application:

- Compile the application using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

- Compile the application using the **-Wlp64** compiler command option.
- Link the application to the **yputdll** library (**/G/system/zdll***nnn***/yputdll**).

For detailed information about writing multi-threaded and 64-bit applications for the Open System Services environment, see the *Open System Services Programmer's Guide*.

**RETURN VALUES**

Upon successful completion, the **writev( )** function returns the number of bytes that were actually written. Otherwise, the value -1 is returned, and **errno** is set to indicate the error.

**ERRORS**

If any of these conditions occurs, the **writev( )** function sets **errno** to the corresponding value:

[EAGAIN] One of these conditions occurred:

- An attempt was made to write to a file descriptor that cannot accept data, and the **O_NONBLOCK** flag is set.
- A write to a pipe (FIFO file) of **PIPE_BUF** bytes or less is requested, **O_NONBLOCK** is set, and not enough free space is available.
- The **O_NONBLOCK** flag is set on this file, and the process would be delayed in the write operation.

[EALREADY] Operation already in progress. An I/O operation started by a thread-aware function is in progress on a regular file and a function that is process-blocking for regular files attempts to begin an I/O operation on the same open file.

If the **writev( )** function is thread-aware, the [EALREADY] value is not returned.

[EBADF] The *filedes* parameter is not a valid file descriptor open for writing.

[ECONNRESET]
One of these conditions occurred:

- The transport-provider process for this socket is no longer available.
- The TCP/IP subsystem for this socket is no longer available.
- The connection was forcibly closed by the peer socket.

The file descriptor specified by the *filedes* parameter can only be closed.

[EFAULT] Part of the *iov* parameter points to a location outside of the allocated address space of the process.

[EFBIG] The application is attempting to write at or beyond the file offset maximum established when the file was opened.

[EGUARDIANLOCKED]
A **writev( )** operation was attempted to a file in the Guardian file system (that is, a file in **/G**) that is locked.

[EINTR] A **writev( )** operation was interrupted by a signal before any data was written.

[EINVAL] One of these conditions occurred:

- The file position pointer associated with the file specified by the *filedes* parameter was negative.

- The value of the *iov_count* parameter was less than or equal to 0 (zero), or greater than **IOV_MAX**.
- One of the **iov_len** values in the *iov* array was negative or overflowed a data item of type **ssize_t**.
- The sum of the **iov_len** values in the *iov* array overflowed an integer.

[EIO] One of these conditions occurred:

- The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
- A physical I/O error occurred. The device holding the file might be in the down state, or both processors that provide access to the device might have failed. Data might have been lost during a transfer.

[EISGUARDIAN]
The value used for the *filedes* parameter is appropriate only in the Guardian environment.

[ENETDOWN]
The *filedes* parameter specifies a file on a remote HP NonStop node, but communication with the remote node has been lost.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOSPC] No free space is left on the fileset containing the file.

[ENOTCONN] An attempt was made to write to a socket that is not connected to a peer socket.

[ENXIO] One of these conditions occurred:

- The device associated with the file descriptor specified by the *filedes* parameter is a block special device or character special file, and the file pointer is out of range.
- No existing device is associated with the file descriptor specified by the *filedes* parameter.

[EPIPE] One of these conditions occurred:

- An attempt was made to write to a pipe or FIFO file that is not open for reading by any process. A **SIGPIPE** signal is sent if the process is running in the OSS environment.
- An attempt was made to write to a pipe that has only one end open.
- An attempt was made to write to a socket that is shut down or closed.

[ETIMEDOUT]
Data transmission on the socket timed out.

[EWOULDBLOCK]
The process attempted an operation on a socket for which **O_NONBLOCK** is set, there is no space available, and no error has occurred.

[EWRONGID] One of these conditions occurred:

- The process attempted an input or output operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation, and takeover by the backup process occurred.
- The open file descriptor has migrated to a new processor but the new processor lacks a resource or system process needed for use of the file descriptor.

The file descriptor specified by the *filedes* parameter can only be closed.

For all other error conditions, **errno** is set to the appropriate Guardian file-system error number. See the *Guardian Procedure Errors and Messages Manual* for more information about a specific Guardian file-system error.

## RELATED INFORMATION

Functions: **creat(2)**, **creat64(2)**, **fcntl(2)**, **lseek(2)**, **lseek64(2)**, **open(2)**, **open64(2)**, **pipe(2)**, **socket(2)**, **spt_writevz(2)**, **ulimit(3)**.

## STANDARDS CONFORMANCE

The HP implementation does not generate the **SIGXFSZ** signal.

HP extensions to the XPG4 Version 2 specification are:

- The **errno** values [ECONNRESET], [EFAULT], [EGUARDIANLOCKED], [EINVAL], [ENETDOWN], [ENOTCONN], [ETIMEDOUT], and [EWRONGID] can be returned.
- For systems running J06.07 and later J-series RVUs or H06.18 and later H-series RVUs, the **errno** value [ENOMEM] can be returned when there is not enough system memory available to complete the operation.

The use of this function with the POSIX User Thread Model library conforms to industry standards as follows:

- IEEE Std 1003.1-2004, POSIX System Application Program Interface
- When a signal arrives during a call to a thread-aware **writev( )** function, the thread-aware **writev( )** retries the I/O operation instead of returning the **errno** value [EINTR] with the following exception. If the thread-aware **fork( )** function is called by a signal handler that is running on a thread performing a thread-aware **writev( )** call, the thread-aware **writev( )** call in the child process returns [EINTR] to the application.

# Section 11. Files

This section contains reference pages for some Open System Services (OSS) header files and special files. These reference pages reside in the **cat4** and **cat7** directories and are sorted alphabetically by U.S. English conventions in this section.

## NAME

**ar -** Describes the archive (library) file format

## SYNOPSIS

**#include <ar.h>**

## DESCRIPTION

The **ar** archive command combines several files into one. Archives are used mainly as libraries to be searched by the Binder utility for TNS or accelerated programs and by the **nld** utility for TSN/R native programs.

A file produced by the **ar** command has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number and header layout are described in the **ar.h** header file.

Each file begins on an even boundary. A newline character is inserted between files if necessary; nevertheless, the size given reflects the actual size of the file exclusive of padding.

There is no provision for empty areas in an archive file.

The encoding of the header is portable across machines. If an archive contains printable files, the archive itself is printable.

The header is made up of six fixed-length ASCII fields, followed by a 2-character trailer. The fields are as follows:

| | |
|---|---|
| **ar_name** | Object name (16 characters) |
| **ar_date** | File's last modification time (12 characters); for example, UTC seconds since the Epoch |
| **ar_uid** | User ID (6 characters) |
| **ar_gid** | Group ID (6 characters) |
| **ar_mode** | File's mode (8 characters) |
| **ar_size** | File's size (10 characters) |

All numeric fields are in decimal, except for the file mode, which is in octal.

The 2-byte trailer is the string

```
`\n
```

Only the **ar_name** field provides for overflows. If any filename is more than 16 characters in length or contains an embedded space, the string

```
#1/
```

followed by the ASCII length of the filename, is written in the **ar_name** field. The file size (stored in the archive header) is incremented by the length of the filename. The filename is then written immediately following the archive header.

## RELATED INFORMATION

Commands: **ar(1)**, **nm(1)**.

## NAME

**core**, **saveabend -** Is a file containing a memory image

## DESCRIPTION

In the OSS implementation, the equivalent of a core file is a saveabend file. A saveabend file is a type of process snapshot file and can be used with a system debugger when necessary.

A saveabend file is created whenever a process terminates abnormally and it is possible to create a saveabend file. The location of a saveabend file is displayed on the home terminal of the terminated process.

A saveabend file is created according to the following naming convention:

**ZZSA***nnnn*

where *nnnn* is a numeric increment.

The saveabend file contains data-area and file-status information at the time of the failure. Use a symbolic debugger program to examine the saveabend file. Refer to the appropriate manual for the symbolic debugger for information on saveabend files and how to examine them.

## NOTES

There is no **core.h** header file.

Saveabend files are sometimes referred to as Guardian save or SAVEABEND files.

## RELATED INFORMATION

Commands: **c89(1)**, **gtacl(1)**, **osh(1)**, **run(1)**, **runv(1)**, **sh(1)**.

Functions: **sigaction(2)**, **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**.

Files: **signal(4)**.

## NAME

**cpio -** Describes the extended **cpio** archive file format

## SYNOPSIS

**#include <cpio.h>**

## DESCRIPTION

The byte-oriented **cpio** archive file format is a series of entries, each entry made up of a header that describes the file and the name of the file, followed by the contents of the file.

The format of the **cpio** header is described below.

Table 11−1. cpio Archive File Header Format

| Header Field Name | Length (in octets) | Interpretation | Contents |
|---|---|---|---|
| **c_magic** | 6 | Octal number | Identifies the archive as transportable by containing the value 070707 |
| **c_dev** | 6 | Octal number | ID of the device containing the file |
| **c_ino** | 6 | Octal number | File serial number |
| **c_mode** | 6 | Octal number | File type and access permission |
| **c_uid** | 6 | Octal number | User ID of the owner |
| **c_gid** | 6 | Octal number | Group ID of the owner |
| **c_nlink** | 6 | Octal number | Number of links referencing the file at the time the archive was created |
| **c_rdev** | 6 | Octal number | Not used |
| **c_mtime** | 11 | Octal number | Latest modification time of the file at the time the archive was created |
| **c_namesize** | 6 | Octal number | Length of the pathname, including the terminating null character |
| **c_filesize** | 11 | Octal number | Byte length of the file data |

The archive entry for the name of a file has the following format:

Table 11–2. cpio Archive File Filename Entry Format

| **Field Name** | **Length (in octets)** | **Interpretation** |
|---|---|---|
| **c_name** | **c_namesize** | Pathname string |

The **c_name** field contains the pathname of the file as a string with the length given by the **c_namesize** field in the file header. This string length includes the null character that terminates the name. If the filename is found on a medium that would create an invalid pathname, the **pax** utility skips the file and displays an error message to the standard error file.

Following the header and the pathname string, the **cpio** archive file data has the following form:

Table 11–3. cpio Archive File Data Format

| **Field Name** | **Length (in octets)** | **Interpretation** |
|---|---|---|
| **c_filedata** | **c_filesize** | Data |

If the **c_filesize** field of the header has the value 0 (zero), then the file is empty.

A header denoting the filename **TRAILER!!!** indicates the end of the archive; what follows this header is undefined.

Only regular files contain data that can be restored. FIFO special files, directories, and the trailer are archived with the **c_filesize** field of the header equal to 0 (zero); these objects are restored with the **pax** utility as directories and FIFOs.

The **cpio.h** header file contains the following macro definitions:

Table 11−4. cpio.h Header File Macros

| Macro | Value (in octal) | Interpretation |
|---|---|---|
| **C_IRUSR** | 0000400 | Read by owner |
| **C_IWUSR** | 0000200 | Write by owner |
| **C_IXUSR** | 0000100 | Execute by owner |
| **C_IRGRP** | 0000040 | Read by group |
| **C_IWGRP** | 0000020 | Write by group |
| **C_IXGRP** | 0000010 | Execute by group |
| **C_IROTH** | 0000004 | Read by others |
| **C_IWOTH** | 0000002 | Write by others |
| **C_IXOTH** | 0000001 | Execute by others |
| **C_ISUID** | 0004000 | Set user ID |
| **C_ISGID** | 0002000 | Set group ID |
| **C_ISVTX** | 0001000 | Reserved |
| **C_ISDIR** | 0040000 | Directory |
| **C_ISFIFO** | 0010000 | FIFO |
| **C_ISREG** | 0100000 | Regular file |
| **C_ISBLK** | 0060000 | Block special file |
| **C_ISCHR** | 0020000 | Character special file |
| **C_ISCTG** | 0110000 | Reserved |
| **C_ISLNK** | 0120000 | Reserved |
| **C_ISSOCK** | 0140000 | Reserved |
| **MAGIC** | 070707 | Magic number value |

## RELATED INFORMATION

Commands: **pax(1)**, **pinstall(1)**.

## NAME

**dir -** Describes the format of directories

## SYNOPSIS

**#include <sys/types.h>**
**#include <dirent.h>**

## DESCRIPTION

A directory is a file that contains directory entries. The fact that a file is a directory is indicated by a bit in the flag word of the inode entry for the file.

Users cannot write a directory. Users can read directory entries by making calls to the **readdir( )** function after opening the directory file by calling the **opendir( )** function.

Directory entries are returned in directory entry structures, which are of variable length. Each directory entry has a **dirent** structure at the beginning, containing its inode number, the length of the entry, and the length of the filename contained in the entry. These structure components are followed by the filename, padded to a 4-byte boundary with null bytes. All names are guaranteed null-terminated. The maximum permitted length of a name in a directory can be obtained by calling the **pathconf()** function.

By convention, the first two entries in each directory are for **.** (dot) and **..** (dot-dot). The **.** (dot) entry is for the directory itself. The **..** (dot-dot) entry is for the parent directory. The meaning of **..** (dot-dot) is modified for the **/** (root) directory of the OSS file system, where **..** (dot-dot) has the same meaning as **.** (dot).

Guardian directories (that is, directories within the **/G** file system) behave the same as OSS directories, with the following exceptions:

- The **mkdir( )** function successfully creates a directory within **/G** only when all of the following are true:
  - — The directory is exactly three directories under the root (for example, **/G/vol/subvol**).
  - — The filenames in the directory pathname use correct Guardian naming syntax (otherwise, **errno** is set to [EINVAL]).
  - — The directory corresponds to a Guardian subvolume.
- The **mkdir( )** function succeeds on a directory that is an existing Guardian subvolume (for example, **/G/vol/subvol**) only when the Guardian subvolume is empty. If the Guardian subvolume is not empty, the **mkdir( )** function fails and **errno** is set to the value [EEXIST].

## RELATED INFORMATION

Functions: **chdir(2)**, **closedir(3)**, **mkdir(2)**, **opendir(3)**, **readdir(3)**, **rewinddir(3)**, **rmdir(2)**.

## NAME

**float -** Specifies the system limits for floating-point operations

## SYNOPSIS

**#include <float.h>**

## DESCRIPTION

The **float.h** header file defines symbolic names. These symbolic names represent floating-point values for the two possible floating-point formats that a program can use. The floating-point format is chosen at compilation time.

See the **float.h** header file for the actual values of these limits in the HP implementation.

The values shown in the following table depend on whether the process is using Tandem-format floating-point data (using the **-WTandem_float c89** compiler flag or equivalent) or is using IEEE floating-point data (using the **-WIEEE_float c89** compiler flag or equivalent).

Table 11–5. Values for Floating-Point Constants

| Symbolic Constant | Tandem-Format Value | IEEE-Format Value |
|---|---|---|
| **DBL_DIG** | 16 | 15 |
| **DBL_EPSILON** | 5.551115123125782720e-17 | 2.2204460492503131E-16 |
| **DBL_MANT_DIG** | 55 | 53 |
| **DBL_MAX** | 1.15792089237316192e77 | 1.7976931348623157E+308 |
| **DBL_MAX_EXP** | 256 | 1024 |
| **DBL_MAX_10_EXP** | 77 | 308 |
| **DBL_MIN** | 1.7272337110188889e-77 | 2.2250738585072014E-308 |
| **DBL_MIN_EXP** | -254 | -1021 |
| **DBL_MIN_10_EXP** | -77 | -307 |
| **FLT_EPSILON** | 2.3841858e-17 | 1.19209290E-07F |
| **FLT_MANT_DIG** | 23 | 24 |
| **FLT_MAX** | 1.1579208e77F | 3.40282347E+38F |
| **FLT_MAX_EXP** | 256 | 128 |
| **FLT_MAX_10_EXP** | 77 | 38 |
| **FLT_MIN** | 1.7272337e-77F | 1.17549435E-38F |
| **FLT_MIN_EXP** | -254 | -125 |
| **FLT_MIN_10_EXP** | -77 | 37 |
| **LDBL_DIG** | 16 | 15 |
| **LDBL_EPSILON** | 5.551115123125782720e-17 | 2.2204460492503131E-16 |
| **LDBL_MANT_DIG** | 55 | 53 |
| **LDBL_MAX** | 1.15792089237316192e77 | 1.7976931348623157E+308 |
| **LDBL_MAX_EXP** | 256 | 1024 |
| **LDBL_MAX_10_EXP** | 77 | 308 |
| **LDBL_MIN** | 1.7272337110188889e-77 | 2.2250738585072014E-308 |
| **LDBL_MIN_EXP** | -254 | -1021 |
| **LDBL_MIN_10_EXP** | -77 | -307 |

## RELATED INFORMATION

Files: **limits(4)**.

## STANDARDS CONFORMANCE

This file conforms to the XPG4 version 2 specification when used for IEEE floating-point data.

Remember the following rules when using any special floating-point mode:

- Do not assume that functions such as **printf( )** or **tanh( )** behave correctly if you call them after setting a nondefault mode (such as rounding toward zero). Unless a function is documented as tolerating such settings, you should restore the default operating mode before calling the function.
- The exception bits of the status register stay set until they are explicitly cleared.

## NAME

**limits -** Specifies the system limits

## SYNOPSIS

**#include <limits.h>**

## DESCRIPTION

The **limits.h** header file defines symbolic names. These symbolic names represent:

- Implementation-dependent constants whose values set limits on system resources used by applications in the OSS environment. These values are all at least as large as minimum acceptable values set by the POSIX.1, POSIX.2, XPG4, and IEEE Std 1003.1-2004 standards. See **Implementation-Dependent Constants**, later in this reference page.
- POSIX.1, POSIX.2 and IEEE Std 1003.1-2004 standard minimum acceptable values. See **POSIX-Defined Minimum Values**, later.
- Floating-point values for the two possible floating-point formats that a program can use. The floating-point format is chosen at compilation time.

See the **limits.h** header file for the actual values of these limits in the HP implementation.

Some of the implementation-dependent constants have values that can increase at run time. These runtime values can be determined at run time using the **sysconf( )** function.

Other limiting values are available only at run time for one of the following reasons:

- The limit is pathname-dependent.
- The limit differs between compile time and run time.

These values are not specified in the **limits.h** header file. For completeness, they are listed under **Values Unknown at Compile Time**, later in this reference page.

An application can use the **fpathconf( )**, **pathconf( )**, and **sysconf( )** functions to determine the actual value of any limit at run time.

### Implementation-Dependent Constants

The following values are defined in the **limits.h** header file. Some of these values are minimum values that can increase at run time. Such values are indicated as "runtime-increasable."

**BC_BASE_MAX**

Maximum **obase** value allowed by the **bc** utility. This is a runtime-increasable value. Use the **sysconf( )** function to obtain the runtime value.

**BC_DIM_MAX**

Maximum number of elements permitted in an array by the **bc** utility. This is a runtime-increasable value. Use the **sysconf( )** function to obtain the runtime value.

**BC_SCALE_MAX**

Maximum **scale** value allowed by the **bc** utility. This is a runtime-increasable value. Use the **sysconf( )** function to obtain the runtime value.

**BC_STRING_MAX**

Maximum length of a string constant accepted the **bc** utility. This is a runtime-increasable value. Use the **sysconf( )** function to obtain the runtime value.

**CHAR_BIT** Number of bits in an object of type **char**. This value is always 8.

**CHARCLASS_NAME_MAX**
Maximum number of bytes in a character class name. This value is always 255.

**CHAR_MAX** Maximum value for a signed **char**. In the HP implementation, the type **char** is not considered a signed integer; **CHAR_MAX** is therefore treated like **UCHAR_MAX**.

**CHAR_MIN** Minimum value for a signed **char**. In the HP implementation, the type **char** is not considered a signed integer; **CHAR_MIN** is therefore 0 (zero).

**COLL_WEIGHTS_MAX**
Maximum number of weights that can be assigned to an entry of the **LC_COLLATE** order keyword in the locale definition file. This is a runtime-increasable value. Use the **sysconf( )** function to obtain the runtime value.

**EXPR_NEST_MAX**
Maximum number of expressions that can be nested within parentheses by the **expr** utility. This is a runtime-increasable value. Use the **sysconf( )** function to obtain the runtime value.

**INT_BIT** Number of bits in an object of type **int**. In the HP implementation, this value is 32.

**INT_MAX** Maximum value for an object of type **int**. This value depends on the size of an integer, which, in the HP implementation, is 32 bits.

**INT_MIN** Minimum value for an object of type **int**. This value depends on the size of an integer, which, in the HP implementation, is 32 bits.

**LINE_MAX** Unless otherwise noted, the maximum length, in bytes, of the input line to a utility (from either the standard input file or another file), when the utility is described as processing text files. The length includes room for the trailing newline character. This is a runtime-increasable value. Use the **sysconf( )** function to obtain the runtime value.

**LLONG_BIT** Number of bits in an object of type **long long int**. This symbolic constant is specific to the HP implementation.

**LLONG_MAX**
Maximum value for an object of type **long long int**. This symbolic constant is specific to the HP implementation.

**LLONG_MIN** Minimum value for an object of type **long long int**. This symbolic constant is specific to the HP implementation.

**LOGIN_NAME_MAX**
Maximum length of a user or alias name in calls by the **getlogin_r( )** function.

This define is supported for systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs only.

**LONG_BIT** Number of bits in an object of type **long int**.

**LONG_MAX** Maximum value for an object of type **long int**.

**LONG_MIN** Minimum value for an object of type **long int**.

**MB_LEN_MAX**
Maximum number of bytes in a character for any supported locale.

**NL_ARGMAX**
Maximum value of the *digit* parameter in calls to the **printf( )** and **scanf( )** functions.

**NL_MSGMAX**
Maximum message number.

**NL_NMAX** Maximum number of bytes in an N-to-1 collation mapping.

**NL_SETMAX** Maximum number of filesets per catalog.

**NL_TEXTMAX**
Maximum number of bytes in a message string.

**PATH_MAX** Maximum number of bytes in a pathname including the terminating null character.

**PIPE_BUF** Maximum number of bytes that is guaranteed to be transferred as a unit when writing to a pipe.

**RE_DUP_MAX**
Maximum number of repeated occurrences of a regular expression permitted when using the *m***,***n* interval notation. This is a runtime-increasable value. Use the **sysconf( )** function to obtain the runtime value.

**SCHAR_MAX**
Maximum value for an object of type **signed char**.

**SCHAR_MIN** Minimum value for an object of type **signed char**.

**SHRT_MAX** Maximum value for an object of type **short**.

**SHRT_MIN** Minimum value for an object of type **short**.

**TZNAME_MAX**
Maximum number of bytes supported for the name of a time zone (not of the **TZ** variable).

**UCHAR_MAX**
Maximum value for an object of type **unsigned char**.

**UINT_MAX** Maximum value for an object of type **unsigned int**. This value depends on the size of an integer, which, in the HP implementation, is 32 bits.

**ULLONG_MAX**
Maximum value for an object of type **unsigned long long int**. This symbolic constant is specific to the HP implementation.

**ULONG_MAX**
Maximum value for an object of type **unsigned long int**.

**USHRT_MAX**
Maximum value for an object of type **unsigned short int**.

**WORD_BIT** Number of bits in a word of type **int**.

### POSIX-Defined Runtime Invariant Values (Possibly Indeterminate)

The symbolic constants in the following list are defined in the **limits.h** header file if the application has been compiled using the **_PUT_MODEL_** feature test macro or equivalent compiler command option. These symbolic names represent the maximum value for certain features. The indetermination of these values might depend on the amount of available memory space on a specific instance of a specific implementation. The actual value supported by a specific instance is provided by the **sysconf( )** function.

These constants are supported for systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs only.

**PTHREAD_DESTRUCTOR_ITERATIONS**
Maximum number of attempts made to destroy a thread's thread-specific data values on thread exit.

**PTHREAD_KEYS_MAX**
Maximum number of data keys that can be created by a process.

**PTHREAD_STACK_MIN**
Maximum size in bytes of thread stack storage.

**PTHREAD_THREADS_MAX**
Maximum number of threads that can be created per process.

### POSIX-Defined Minimum Values

The symbolic constants in the following list are defined in the **limits.h** header file with values specified by the POSIX.1, POSIX.2, POSIX.12 draft, or IEEE Std 1003.1-2004 standards. These symbolic names represent the most restrictive value for certain features. A portable application must not require a larger value for correct operation. The HP implementation defines some of the related symbolic constants to be less restrictive.

These values are the same in all POSIX-compliant implementations.

**_POSIX_ARG_MAX**
Maximum length in bytes of argument data to a function in the **exec** and **tdm_exec** sets of functions, including environment data.

**_POSIX_CHILD_MAX**
Maximum number of simultaneous processes per real user ID.

**_POSIX_FD_SETSIZE**
Maximum number of file descriptors that the process can use with the **select( )** function.

**_POSIX_HIWAT**
Maximum number of bytes that a process can buffer on a socket for a send or receive action.

**_POSIX_LINK_MAX**
Maximum number of links to a single file.

**_POSIX_MAX_CANON**
Maximum number of bytes in a terminal canonical input queue.

**_POSIX_MAX_INPUT**
Maximum number of bytes allowed in a terminal input queue.

**_POSIX_NAME_MAX**
Maximum number of bytes in a filename excluding the terminating null.

**_POSIX_NGROUPS_MAX**
Maximum number of simultaneous supplementary group IDs per process.

**_POSIX_OPEN_MAX**
Maximum number of files that a process can have open at a time.

**_POSIX_PATH_MAX**
Maximum number of bytes in a pathname including the terminating null.

**_POSIX_PIPE_BUF**
Maximum number of bytes that is guaranteed to be atomic when writing to a pipe.

**_POSIX_QLIMIT**
Maximum number of connections that the process can queue on a single socket.

**_POSIX_SSIZE_MAX**
Maximum value that can be stored in an object of type **ssize_t**.

**_POSIX_STREAM_MAX**
Maximum number of streams that one process can have open at one time.

**_POSIX_THREAD_DESTRUCTOR_ITERATIONS**
Minimum number of attempts made to destroy a thread's thread-specific data values on thread exit.

This define is supported for systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs only and requires that the application has been compiled using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

**_POSIX_THREAD_KEYS_MAX**
Maximum number of data keys per process.

This define is supported for systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs only and requires that the application has been compiled using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

**_POSIX_THREAD_THREADS_MAX**
Maximum number of threads per process.

This define is supported for systems running H06.21 or later H-series RVUs or J06.10 or later J-series RVUs only and requires that the application has been compiled using the **_PUT_MODEL_** feature test macro or equivalent compiler command option.

**_POSIX_TZNAME_MAX**
Maximum number of bytes supported for the name of a time zone (not of the **TZ** variable).

**_POSIX2_BC_BASE_MAX**
Maximum **obase** values allowed by the **bc** utility.

**_POSIX2_BC_DIM_MAX**
Maximum number of elements permitted in an array by the **bc** utility.

**_POSIX2_BC_SCALE_MAX**
Maximum **scale** value allowed by the **bc** utility.

**_POSIX2_BC_STRING_MAX**
Maximum length of a string constant accepted by the **bc** utility.

**_POSIX2_COLL_WEIGHTS_MAX**
Maximum number of weights that can be assigned to an entry of the **LC_COLLATE** order keyword in the locale definition file.

**_POSIX2_EXPR_NEST_MAX**
Maximum number of expressions that can be nested within parentheses by the **expr** utility.

**_POSIX2_LINE_MAX**
Unless otherwise noted, the maximum length, in bytes, of the input line to a utility (from either the standard input file or another file), when the utility is described as processing text files. The length includes room for the trailing newline character.

**_POSIX2_RE_DUP_MAX**
Maximum number of repeated occurrences of a regular expression permitted when using the interval notation *m***,***n*.

**Values Unknown at Compile Time**

The following values are unknown at compile time and are therefore not defined in the **limits.h** header file:

**ARG_MAX** Maximum length in bytes of argument data to a function in the **exec**, **tdm_exec**, and **tdm_spawn** sets of functions. Use the **sysconf( )** function to obtain this value at run time.

**CHILD_MAX** Maximum number of simultaneous processes per real user ID. Use the **sysconf( )** function to obtain this value at run time.

**IOV_MAX** Maximum number of **iovec** structures that a process can use at a given time for scatter or gather operations. Use the **sysconf( )** function to obtain this value at run time.

**LINK_MAX** Maximum value of a file's link count. Use the **pathconf( )** function to obtain this value at run time.

**MAX_CANON**
Maximum number of bytes in a terminal canonical input line. Use the **pathconf( )** function to obtain this value at run time.

**MAX_INPUT** Minimum number of bytes for which space is available in a terminal input queue; therefore, the maximum number of bytes a portable application can require to be entered as input before it reads them. Use the **pathconf( )** function to obtain this value at run time.

**NAME_MAX** Maximum number of bytes in a filename (excluding the terminating null). Use the **pathconf( )** function to obtain this value at run time.

**NL_LANGMAX**
Maximum number of bytes in a **LANG** name. Use the **pathconf( )** function to obtain this value at run time.

**OPEN_MAX** Maximum number of files that one process can have open at any one time. Use the **sysconf( )** function to obtain this value at run time.

**SOCK_MAXBUF**
Maximum number of bytes in a buffer to be used with a socket. Use the **sysconf( )** function to obtain this value at run time.

**STREAM_MAX**
The number of streams that one process can have open at one time. Use the **sysconf( )** function to obtain this value at run time.

### Floating-Point Values

The values shown in the following table depend on whether the process is using HP floating-point data (using the **-WTandem_float** C compiler flag or equivalent) or is using IEEE floating-point data (using the **-WIEEE_float** C compiler flag or equivalent).

Table 11−6. Values for Floating-Point Constants

| Symbolic Constant | Tandem-Format Value | IEEE-Format Value |
|---|---|---|
| **DBL_DIG** | 16 | 15 |
| **DBL_MAX** | 1.15792089237316192e77 | 1.7976931348623157E+308 |
| **FLT_MAX** | 1.1579208e77F | 3.40282347E+38F |

## RELATED INFORMATION

Functions: **fpathconf(3)**, **pathconf(3)**, **sysconf(3)**.

Files: **float(4)**.

## STANDARDS CONFORMANCE

The HP implementation does not define **ATEXIT_MAX**, **NZERO**, **PAGE_SIZE**, **PAGESIZE**, or **PASS_MAX**.

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- The following symbolic constants are assigned values that define limits above the maximum or below the minimum required by the XPG4 Version 2, POSIX.1, POSIX.2, and IEEE Std 1003.1-2004 standards:

The XPG4 version 2 specification indicates that the floating-point symbolic constants **DBL_DIG**, **DBL_MAX**, and **FLT_MAX** are to be withdrawn from the specification.

Table 11−7. Values for Symbolic Constants

| Symbolic Constant | POSIX-Defined Value | OSS Value |
|---|---|---|
| **CHARCLASS_NAME_MAX** | 14 | 255 |
| **CHAR_MAX** | **SCHAR_MAX** (127) or **UCHAR_MAX** (255) | **UCHAR_MAX** (255) |
| **CHAR_MIN** | **SCHAR_MIN** (127) or 0 (zero) | 0 (zero) |
| **COLL_WEIGHTS_MAX** | 2 | 6 |
| **INT_MAX** | 32767 | 2147483647 |
| **INT_MIN** | -32767 | -2147483647 |
| **LONG_MIN** | -2147483647 | -2147483648 |
| **MB_LEN_MAX** | 1 | 4 |
| **NGROUPS_MAX** | 0 | 32 |
| **NL_MSGMAX** | 32767 | 65535 |
| **NL_NMAX** | No guaranteed minimum | 10 |
| **NL_SETMAX** | 255 | 65535 |
| **NL_TEXTMAX** | **_POSIX_LINE_MAX** (2048) | 8192 |
| **PATH_MAX** | 255 | 1024 |
| **PIPE_BUF** | 512 | 4096 |
| **SCHAR_MIN** | -127 | -128 |
| **SHRT_MIN** | -32767 | -32768 |
| **SSIZE_MAX** | 32767 | 53248 |
| **UINT_MAX** | 65535 | 4294967295 |

The following are HP extensions to the XPG4 Version 2 specification:

- The values **INT_BIT**, **LLONG_BIT**, **LLONG_MAX**, and **LLONG_MIN**.
- The sockets-related values **_POSIX_FD_SETSIZE**, **_POSIX_HIWAT**, **_POSIX_QLIMIT**, and **SOCK_MAXBUF**.

## NAME

**math -** Specifies mathematical functions, constants, and types

## SYNOPSIS

**#include <math.h>**

## DESCRIPTION

The **math.h** header file defines the following types:

**float_t** Specifies a floating type at least as wide as **float**.

**double_t** Specifies a floating type at least as wide as **double**.

The following macros are defined, where *floating-type* indicates an expression of the float type:

**int fpclassify(**
*floating-type x***);**

**int isfinite(**
*floating-type x***);**

**int isgreater(**
*floating-type x***,**
*floating-type y***);**

**int isgreaterequal(**
*floating-type x***,**
*floating-type y***);**

**int isinf(**
*floating-type x***);**

**int isless(**
*floating-type x***,**
*floating-type y***);**

**int islessequal(**
*floating-type x***,**
*floating-type y***);**

**int islessgreater(**
*floating-type x***,**
*floating-type y***);**

**int isnan(**
*floating-type x***);**

**int isnormal(**
*floating-type x***);**

**int isunordered(**
*floating-type x***,**
*floating-type y***);**

**int signbit(**
*floating-type x***);**

The following constants of type **double** are defined. These constants are accurate within the precision of the **double** type:

**M_E** Specifies the value of *e*.

**M_LN2** Specifies the value of log to the base *e* of 2.

**M_LN10** Specifies the value of log to the base *e* of 10.

**M_LOG2E** Specifies the value of log to the base 2 of *e*.

**M_LOG10E** Specifies the value of log to the base 10 of *e*.

**M_PI** Specifies the value of pi.

**M_PI_2** Specifies the value of pi divided by 2.

**M_PI_4** Specifies the value of pi divided by 4.

**M_SQRT2** Specifies the value of the square root of 2.

**M_SQRT1_2** Specifies the value of 1 divided by the square root of 2.

**M_1_PI** Specifies the value of 1 divided by pi.

**M_2_PI** Specifies the value of 2 divided by pi.

**M_2_SQRTPI** Specifies the value of 2 divided by the square root of pi.

The following symbolic constants are defined:

**HUGE_VAL** Specifies a positive **double** expression that cannot necessarily be represented as a type **float** value. Used as an error indicator when returned as a value for mathematics library functions.

For Tandem-format floating-point data, the value of **HUGE_VAL** is **DBL_MAX**. For IEEE floating-point data, the value of **HUGE_VAL** is positive infinity of type **double**.

**HUGE_VALF** Specifies a positive **float** expression that cannot necessarily be represented as a type **float** value. Used as an error indicator when returned as a value for mathematics library functions.

For Tandem-format floating-point data, the value of **HUGE_VALF** is the same as the value of **FLT_MAX**. For IEEE floating-point data, the value of **HUGE_VALF** is positive infinity of type **float**.

**HUGE_VALL** For IEEE floating-point environments only, specifies a positive **long double** expression. Used as an error indicator when returned as a value for mathematics library functions.

**INFINITY** For IEEE floating-point environments only, specifies a constant expression of type **float** representing positive or unsigned infinity, if available; else to a postive constant of type **float** that overflows at translation time.

**MAXFLOAT** Specifies the value of the maximum noninfinite single-precision floating-point number.

For Tandem-format floating-point data, the value of **MAXFLOAT** is the same as the value of **DBL_MAX**. For IEEE floating-point data, the value of **MAXFLOAT** is also **DBL_MAX**, but the value of **DBL_MAX** differs between the two formats. Refer to the **float(4)** reference page for more information about **DBL_MAX**.

**NAN** For IEEE floating-point environments only, specifies a constant expression of type **float** representing a quiet not-a-number (NaN).

The following macros are defined for number classification. They represent mutually-exclusive kinds of floating-point values and expand to integer constant expressions with distinct values:

**FP_INFINITE**
Infinity.

**FP_NAN** NaN.

**FP_NORMAL** Normalized.

**FP_SUBNORMAL**
Denormalized.

**FP_ZERO** Zero.

The following macros expand to integer constant expressions whose values are returned by **ilogb(***x*):

**FP_ILOGB0** Expands to an integer constant expression whose value is returned by **ilogb(***x*) if *x* is zero. The value of **FP_ILOGB0** is {INT_MIN}.

**FP_ILOGBNAN**
Expands to an integer constant expression whose value is returned by **ilogb(***x*) if *x* is NaN. The value of **FP_ILOGBNAN** is either {INT_MIN}.

The following macros are defined to identify the error handling method supported by the **math.h** functions:

**MATH_ERRNO**
Indicates support for the ISO/IEC C99 **errno** specification.

**MATH_ERREXCEPT**
Indicates support for the ISO/IEC exception flag specification.

**math_errhandling**
Defined to be **MATH_ERRNO** in all situations.

## RELATED INFORMATION

Files: **float(4)**.

## STANDARDS CONFORMANCE

This function conforms to the ISO/IEC 9899:1999 standard.

This function conforms to the IEEE Std 1003.1, 2004 Edition.

Support for Tandem floating-point values is an HP extension to the standards.

## NAME

**named.conf -** configuration file for BIND 9 domain name server named

## DESCRIPTION

**named.conf** is the configuration file for the **named** server.

Within the file, directive statements are enclosed in braces and terminated with a semi-colon. Clauses in the statements are also terminated with a semi-colon. The following comment styles are supported:

| | |
|---|---|
| C style | **/* */** |
| C++ style | **//** to end of line |
| UNIX style | **#** to end of line |

### Directives

#### acl

**acl** *string* **{** *address_match_element***; ... };**

#### key

**key** *domain_name* **{**
**algorithm** *string***;**
**secret** *string***;**
**};**

#### masters

**masters** *string* **[ port** *integer* **] {**
**(** *masters* | *ipv4_address* **[port** *integer***]** |
*ipv6_address* **[port** *integer***] ) [ key** *string* **]; ...**
**};**

#### server

**server (** *ipv4_address* | *ipv6_address* **) {**
**bogus** *boolean***;**
**edns** *boolean***;**
**provide-ixfr** *boolean***;**
**request-ixfr** *boolean***;**
**keys** *server_key***;**
**transfers** *integer***;**
**transfer-format ( many-answers | one-answer );**
**transfer-source (** *ipv4_address* **| * )**
**[ port (** *integer* **| * ) ];**
**transfer-source-v6 (** *ipv6_address* **| * )**
**[ port (** *integer* **| * ) ];**

**support-ixfr** *boolean***; // obsolete**
**};**

#### trusted-keys

**trusted-keys {**
*domain_name flags protocol algorithm key***; ...**
**};**

## controls

**controls {**
**inet (** *ipv4_address* **|** *ipv6_address* **| * )**
**[ port (** *integer* **| * ) ]**
**allow {** *address_match_element***; ... }**
**[ keys {** *string***; ... } ];**
**unix** *unsupported***; // not implemented**
**};**

## logging

**logging {**
**channel** *string* **{**
**file** *log_file***;**
**syslog** *optional_facility***;**
**null;**
**stderr;**
**severity** *log_severity***;**
**print-time** *boolean***;**
**print-severity** *boolean***;**
**print-category** *boolean***;**
**};**
**category** *string* **{** *string***; ... };**
**};**

## lwres

**lwres {**
**listen-on [ port** *integer* **] {**
**(** *ipv4_address* **|** *ipv6_address* **) [ port** *integer* **]; ...**
**};**
**view** *string optional_class***;**
**search {** *string***; ... };**
**ndots** *integer***;**
**};**

## options

**options {**
**avoid-v4-udp-ports {** *port***; ... };**
**avoid-v6-udp-ports {** *port***; ... };**
**blackhole {** *address_match_element***; ... };**
**coresize** *size***;**
**datasize** *size***;**
**directory** *quoted_string***;**
**dump-file** *quoted_string***;**
**files** *size***;**
**heartbeat-interval** *integer***;**
**host-statistics** *boolean***; // not implemented**
**hostname (** *quoted_string* **| none );**
**interface-interval** *integer***;**
**listen-on [ port** *integer* **] {** *address_match_element***; ... };**
**listen-on-v6 [ port** *integer* **] {** *address_match_element***; ... };**
**match-mapped-addresses** *boolean***;**
**memstatistics-file** *quoted_string***;**

**pid-file (** *quoted_string* **| none );**
**port** *integer***;**
**querylog** *boolean***;**
**recursing-file** *quoted_string***;**
**random-device** *quoted_string***;**
**recursive-clients** *integer***;**
**serial-query-rate** *integer***;**
**server-id (** *quoted_string* **| none |;**
**stacksize** *size***;**
**statistics-file** *quoted_string***;**
**statistics-interval** *integer***; // not yet implemented**
**tcp-clients** *integer***;**
**tcp-listen-queue** *integer***;**
**tkey-dhkey** *quoted_string integer***;**
**tkey-gssapi-credential** *quoted_string***;**
**tkey-domain** *quoted_string***;**
**transfers-per-ns** *integer***;**
**transfers-in** *integer***;**
**transfers-out** *integer***;**
**use-ixfr** *boolean***;**
**version (** *quoted_string* **| none );**
**allow-recursion {** *address_match_element***; ... };**
**sortlist {** *address_match_element***; ... };**
**topology {** *address_match_element***; ... }; // not implemented**
**auth-nxdomain** *boolean***; // default changed**
**minimal-responses** *boolean***;**
**recursion** *boolean***;**
**rrset-order {**
**[ class** *string* **] [ type** *string* **]**
**[ name** *quoted_string* **] string** *string***; ...**
**};**
**provide-ixfr** *boolean***;**
**request-ixfr** *boolean***;**
**rfc2308-type1** *boolean***; // not yet implemented**
**additional-from-auth** *boolean***;**
**additional-from-cache** *boolean***;**
**query-source** *querysource4***;**
**query-source-v6** *querysource6***;**
**cleaning-interval** *integer***;**
**min-roots** *integer***; // not implemented**
**lame-ttl** *integer***;**
**max-ncache-ttl** *integer***;**
**max-cache-ttl** *integer***;**
**transfer-format ( many-answers | one-answer );**
**max-cache-size** *size_no_default***;**
**check-names ( master | slave | response )**
**( fail | warn | ignore );**
**cache-file** *quoted_string***;**
**suppress-initial-notify** *boolean***; // not yet implemented**
**preferred-glue** *string***;**
**dual-stack-servers [ port** *integer* **] {**
**(** *quoted_string* **[port** *integer***] |**
**ipv4_address [port** *integer***] |**

**ipv6_address [port** *integer***] ); ...**
**}**
**edns-udp-size** *integer***;**
**root-delegation-only [ exclude {** *quoted_string***; ... } ];**
**disable-algorithms** *string* **{** *string***; ... };**
**dnssec-enable** *boolean***;**
**dnssec-lookaside** *string* **trust-anchor** *string***;**
**dnssec-must-be-secure** *string boolean***;**

**dialup** *dialuptype***;**
**ixfr-from-differences** *ixfrdiff***;**

**allow-query {** *address_match_element***; ... };**
**allow-transfer {** *address_match_element***; ... };**
**allow-update-forwarding {** *address_match_element***; ... };**

**notify** *notifytype***;**
**notify-source (** *ipv4_address* **| * ) [ port (** *integer* **| * ) ];**
**notify-source-v6 (** *ipv6_address* **| * ) [ port (** *integer* **| * ) ];**
**also-notify [ port** *integer* **] { (** *ipv4_address* | *ipv6_address* **)**
**[ port** *integer* **]; ... };**
**allow-notify {** *address_match_element***; ... };**

**forward ( first | only );**
**forwarders [ port** *integer* **] {**
**(** *ipv4_address* | *ipv6_address* **) [ port** *integer* **]; ...**
**};**

**max-journal-size** *size_no_default***;**
**max-transfer-time-in** *integer***;**
**max-transfer-time-out** *integer***;**
**max-transfer-idle-in** *integer***;**
**max-transfer-idle-out** *integer***;**
**max-retry-time** *integer***;**
**min-retry-time** *integer***;**
**max-refresh-time** *integer***;**
**min-refresh-time** *integer***;**
**multi-master** *boolean***;**
**sig-validity-interval** *integer***;**

**transfer-source (** *ipv4_address* **| * )**
**[ port (** *integer* **| * ) ];**
**transfer-source-v6 (** *ipv6_address* **| * )**
**[ port (** *integer* **| * ) ];**

**alt-transfer-source (** *ipv4_address* **| * )**
**[ port (** *integer* **| * ) ];**
**alt-transfer-source-v6 (** *ipv6_address* **| * )**
**[ port (** *integer* **| * ) ];**
**use-alt-transfer-source** *boolean***;**

**zone-statistics** *boolean***;**
**key-directory** *quoted_string***;**

**allow-v6-synthesis {** *address_match_element*; **... }; // obsolete**
**deallocate-on-exit** *boolean*; **// obsolete**
**fake-iquery** *boolean*; **// obsolete**
**fetch-glue** *boolean*; **// obsolete**
**has-old-clients** *boolean*; **// obsolete**
**maintain-ixfr-base** *boolean*; **// obsolete**
**max-ixfr-log-size** *size*; **// obsolete**
**multiple-cnames** *boolean*; **// obsolete**
**named-xfer** *quoted_string*; **// obsolete**
**serial-queries** *integer*; **// obsolete**
**treat-cr-as-space** *boolean*; **// obsolete**
**use-id-pool** *boolean*; **// obsolete**
**};**

### view

**view** *string optional_class* **{**
**match-clients {** *address_match_element*; **... };**
**match-destinations {** *address_match_element*; **... };**
**match-recursive-only** *boolean*;

**key** *string* **{**
**algorithm** *string*;
**secret** *string*;
**};**

**zone** *string optional_class* **{**
**};**

**server (** *ipv4_address* | *ipv6_address* **) {**
**};**

**trusted-keys {**
*string integer integer integer quoted_string*; **...**
**};**

**allow-recursion {** *address_match_element*; **... };**
**sortlist {** *address_match_element*; **... };**
**topology {** *address_match_element*; **... }; // not implemented**
**auth-nxdomain** *boolean*; **// default changed**
**minimal-responses** *boolean*;
**recursion** *boolean*;
**rrset-order {**
**[ class** *string* **] [ type** *string* **]**
**[ name** *quoted_string* **]** *string string*; **...**
**};**
**provide-ixfr** *boolean*;
**request-ixfr** *boolean*;
**rfc2308-type1** *boolean*; **// not yet implemented**
**additional-from-auth** *boolean*;
**additional-from-cache** *boolean*;
**query-source** *querysource4*;
**query-source-v6** *querysource6*;
**cleaning-interval** *integer*;

**min-roots** *integer***; // not implemented**
**lame-ttl** *integer***;**
**max-ncache-ttl** *integer***;**
**max-cache-ttl** *integer***;**
**transfer-format ( many-answers | one-answer );**
**max-cache-size** *size_no_default***;**
**check-names ( master | slave | response )**
**( fail | warn | ignore );**
**cache-file** *quoted_string***;**
**suppress-initial-notify** *boolean***; // not yet implemented**
**preferred-glue** *string***;**
**dual-stack-servers [ port** *integer* **] {**
**(** *quoted_string* **[port** *integer***] |**
*ipv4_address* **[port** *integer***] |**
*ipv6_address* **[port** *integer***] ); ...**
**};**
**edns-udp-size** *integer***;**
**root-delegation-only [ exclude {** *quoted_string***; ... } ];**
**disable-algorithms** *string* **{** *string***; ... };**
**dnssec-enable** *boolean***;**
**dnssec-lookaside** *string trust-anchor string***;**

**dnssec-must-be-secure** *string boolean***;**
**dialup** *dialuptype***;**
**ixfr-from-differences** *ixfrdiff***;**

**allow-query {** *address_match_element***; ... };**
**allow-transfer {** *address_match_element***; ... };**
**allow-update-forwarding {** *address_match_element***; ... };**

**notify** *notifytype***;**
**notify-source (** *ipv4_address* **| * ) [ port (** *integer* **| * ) ];**
**notify-source-v6 (** *ipv6_address* **| * ) [ port (** *integer* **| * ) ];**
**also-notify [ port** *integer* **] { (** *ipv4_address* | *ipv6_address* **)**
**[ port** *integer* **]; ... };**
**allow-notify {** *address_match_element***; ... };**

**forward ( first | only );**
**forwarders [ port** *integer* **] {**
**(** *ipv4_address* | *ipv6_address* **) [ port** *integer* **]; ...**
**};**

**max-journal-size** *size_no_default***;**
**max-transfer-time-in** *integer***;**
**max-transfer-time-out** *integer***;**
**max-transfer-idle-in** *integer***;**
**max-transfer-idle-out** *integer***;**
**max-retry-time** *integer***;**
**min-retry-time** *integer***;**
**max-refresh-time** *integer***;**
**min-refresh-time** *integer***;**
**multi-master** *boolean***;**
**sig-validity-interval** *integer***;**

**transfer-source (** *ipv4_address* **| * )**
**[ port (** *integer* **| * ) ];**
**transfer-source-v6 (** *ipv6_address* **| * )**
**[ port (** *integer* **| * ) ];**

**alt-transfer-source (** *ipv4_address* **| * )**
**[ port (** *integer* **| * ) ];**
**alt-transfer-source-v6 (** *ipv6_address* **| * )**
**[ port (** *integer* **| * ) ];**
**use-alt-transfer-source** *boolean***;**

**zone-statistics** *boolean***;**
**key-directory** *quoted_string***;**

**allow-v6-synthesis {** *address_match_element***; ... }; // obsolete**
**fetch-glue** *boolean***; // obsolete**
**maintain-ixfr-base** *boolean***; // obsolete**
**max-ixfr-log-size** *size***; // obsolete**
**};**

## zone

**zone** *string optional_class* **{**
**type ( master | slave | stub | hint |**
**forward | delegation-only );**
**file** *quoted_string***;**

**masters [ port** *integer* **] {**
**(** *masters* **|**
*ipv4_address* **[port** *integer***] |**
*ipv6_address* **[ port** *integer* **] ) [ key** *string* **]; ...**
**};**

**database** *string***;**
**delegation-only** *boolean***;**
**check-names ( fail | warn | ignore );**
**dialup** *dialuptype***;**
**ixfr-from-differences** *boolean***;**

**allow-query {** *address_match_element***; ... };**
**allow-transfer {** *address_match_element***; ... };**
**allow-update {** *address_match_element***; ... };**
**allow-update-forwarding {** *address_match_element***; ... };**
**update-policy {**
**( grant | deny )** *string*
**( name | subdomain | wildcard | self )** *string*
*rrtypelist***; ...**
**};**

**notify** *notifytype***;**
**notify-source (** *ipv4_address* **| * ) [ port (** *integer* **| * ) ];**
**notify-source-v6 (** *ipv6_address* **| * ) [ port (** *integer* **| * ) ];**
**also-notify [ port** *integer* **] { (** *ipv4_address* **|** *ipv6_address* **)**
**[ port** *integer* **]; ... };**

**allow-notify {** *address_match_element***; ... };**

**forward ( first | only );**
**forwarders [ port** *integer* **] {**
**(** *ipv4_address* **|** *ipv6_address* **) [ port** *integer* **]; ...**
**};**

**max-journal-size** *size_no_default***;**
**max-transfer-time-in** *integer***;**
**max-transfer-time-out** *integer***;**
**max-transfer-idle-in** *integer***;**
**max-transfer-idle-out** *integer***;**
**max-retry-time** *integer***;**
**min-retry-time** *integer***;**
**max-refresh-time** *integer***;**
**min-refresh-time** *integer***;**
**multi-master** *boolean***;**
**sig-validity-interval** *integer***;**

**transfer-source (** *ipv4_address* **| * )**
**[ port (** *integer* **| * ) ];**
**transfer-source-v6 (** *ipv6_address* **| * )**
**[ port (** *integer* **| * ) ];**

**alt-transfer-source (** *ipv4_address* **| * )**
**[ port (** *integer* **| * ) ];**
**alt-transfer-source-v6 (** *ipv6_address* **| * )**
**[ port (** *integer* **| * ) ];**
**use-alt-transfer-source** *boolean***;**

**zone-statistics** *boolean***;**
**key-directory** *quoted_string***;**

**ixfr-base** *quoted_string***; // obsolete**
**ixfr-tmp-file** *quoted_string***; // obsolete**
**maintain-ixfr-base** *boolean***; // obsolete**
**max-ixfr-log-size** *size***; // obsolete**
**pubkey** *integer integer integer quoted_string***; // obsolete**
**};**

## FILES

**/etc/named.conf**

Contains the default configuration file for the **named** server.

## RELATED INFORMATION

Commands: **dnssec_named(8)**, **dnssec_rndc(8)**, **named(8)**, **rndc(8)**.

Documents: *BIND 9 Adminstrators Reference Manual*.

## NAME

**null -** Is a data sink file

## SYNOPSIS

**/dev/null**

## DESCRIPTION

Data written on a **null** special file is discarded.

Reads from a **null** special file always return 0 (zero) bytes.

## EXAMPLES

To create a zero-length file using the **cat** command, enter:

```
cat > foo < /dev/null
```

## NAME

**saveabend -** Is a file containing a memory image

## DESCRIPTION

See the **core(4)** reference page.

## NAME

**signal -** Contains definitions and variables used by signal functions

## SYNOPSIS

**#include <signal.h>**

## DESCRIPTION

The **signal.h** header file contains:

- Declarations of symbolic constants used to refer to the signals that occur in the OSS environment.
- Declarations for the **sigset_t** type and the **sigaction** structure. Note that G-series TNS or accelerated processes and all native processes use different declarations for **sigset_t**; all processes using the same **sigset_t** declaration must be of the same process type.
- Declarations of the **stack_t** structure used to define and manipulate the alternate signal stack. This capability is available only on H-series and J-series RVUs.
- Declarations of additional symbolic constants used in signal handling:

**MINSIGSTKSZ**
Indicates the minimum allowable size of the alternate signal stack.

**SA_NOCLDSTOP**
Indicates that the **SIGCHLD** signal should not be generated when a child process stops.

**SA_ONSTACK**
Note that this flag is not supported on NSK systems. If an alternate signal stack is registered and enabled, and if the thread that defined the signal handler is not blocked when the signal is delivered, the signal handler only runs on the alternate signal stack. If the thread is blocked, the signal handler runs on the user stack. However, **SA_ONSTACK** is provided to allow ported POSIX applications to run without change. **SA_ONSTACK** can be used only if the **SA_COMPATABILITY** value is set. However, you should NOT use the **SA_ONSTACK** flag and the **SA_COMPATABILITY** feature test macro in a threaded application that uses the Standard POSIX Threads library. Use of these two options with the Standard POSIX Threads library can result in undefined behavior in the SPT environment.

**SIG_ABORT** Requests that the process terminate abnormally when a specific signal is received.

**SIG_DEBUG** Requests that the process enter the debugger when a specific signal is received.

**SIG_DFL** Requests default signal handling.

**SIG_ERR** Indicates an error condition by reserving a return code for a certain class of functions. Such functions return a pointer to a function that takes an integer as a parameter and returns void.

**SIG_IGN** Requests that signals be ignored.

**SIGSTKSZ** Indicates the default size of the alternate signal stack.

**SS_DISABLE** Indicates that the alternate signal stack is disabled. No signal handling may be launched at the alternate signal stack.

**SS_ONSTACK**
Indicates that the alternate signal stack is active. A signal handler is currently running on the alternate signal stack.

- Declarations of additional symbolic constants used by the **sigprocmask( )** function for handling process signal masks:

**SIG_BLOCK** Requests a union of the current mask and a supplied value.

**SIG_SETMASK**
Creates a mask from the supplied value.

**SIG_UNBLOCK**
Requests a mask of the current mask less the supplied value.

Note: The **MINSIGSTKSZ**, **SA_ONSTACK**, **SIGSTKSZ**, **SS_DISABLE**, and **SS_ONSTACK** symbolic constants are only supported on systems running J06.10 or later RVUs or H06.21 or later RVUs.

### Signal Generation and Delivery

A signal is said to be generated for (or sent to) a process when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration, and any operating system trap condition normally detectable by a TNS or accelerated Guardian process, in addition to terminal activity, or the invocation of the **kill( )** function.

**Note:** Signals cannot be sent to a TNS or accelerated Guardian process.

Each process has an action to be taken in response to each signal defined by the system. A signal is said to be delivered to a process when the appropriate action for the process and signal is taken.

A process can elect to ignore the delivery of some signals, while allowing the system to perform default actions upon the delivery of other signals. The system also allows processes to install process-specific signal-catching functions.

During the time between the generation of a signal and its delivery, the signal is said to be pending. Usually, this interval cannot be detected by an application.

However, a signal can be blocked from delivery to a process. If the action associated with a blocked signal is anything other than to ignore the signal, and if that signal is generated for the process, the signal remains pending until either it is unblocked or the action associated with it is set to ignore the signal. If the action associated with the blocked signal is to ignore the signal, and if that signal is generated for the process, it is unspecified whether the signal is discarded immediately upon generation or remains pending; applications should therefore not depend on whether the signal is discarded or remains pending.

Each process has a signal mask that defines the set of signals currently blocked from delivery to it. The signal mask for a process is initialized to that of its parent. The **sigaction( )**, **sigprocmask( )**, and **sigsuspend( )** functions control the manipulation of the signal mask.

The determination of which action is taken in response to a signal is made at the time the signal is delivered, allowing for any changes since the time of generation. This determination is independent of the means by which the signal was originally generated.

If a subsequent occurrence of a pending signal is generated, it is discarded and only one instance of the same signal remains pending; however, because this action is likely to change in future releases, users should not rely on this behavior. The order in which pending signals are delivered to a process is unspecified and should not be relied upon.

When any stop signal (**SIGSTOP**, **SIGTSTP**, **SIGTTIN**, or **SIGTTOU**) is generated for a process, any pending **SIGCONT** signal for that process is discarded. Conversely, when **SIGCONT** is generated for a process, all pending stop signals for that process are discarded even if these signals are being caught.

When **SIGCONT** is generated for a process that is stopped, the process continues, even if the **SIGCONT** signal is blocked or ignored. If **SIGCONT** is blocked and not ignored, it remains pending until either it is unblocked or a stop signal is generated for the process.

When **SIGUNCP** is generated, the action taken allows an H-series process to control what happens when the process does not offer frequent enough opportunities to synchronize blade elements. Either the process can call a user signal-handler function, or the following actions are defined:

**SIG_ABORT** The process terminates abnormally.

**SIG_DEBUG** The process enters the debugger so that an appropriate location for a voluntary rendezvous opportunity can be identified.

**SIG_DFL** The signal is ignored by the process, but the system suspends the process until an EMS event is generated for logging by the $ZLOG distributor.

**SIG_IGN** The signal is ignored by the process and no EMS event is generated.

When a signal handler is invoked (which normally only happens to allow debugging), the process should call **sigaction( )** to set **SIG_IGN** or **SIG_DFL** or simply return. It is undefined what happens if the signal is rearmed and **siglongjmp( )** is called before the synchronization completes: the process might suspend or it might abend.

Signals reserve the last 1000 words of the user stack. When a **SIGSTK** signal is delivered, the system cuts back the stack to the start of the last 1000 words. If user data is present in the last 1000 words of the stack, that data is lost.

### The Signals

The following table lists each signal name and corresponding signal number and default action. On receipt of one of these signals, the application can elect to:

- Accept the default action; see **Default Action**, later in this reference page.
- Ignore the signal; see **Ignoring a Signal**, later.
- Catch the signal by invoking a signal-specific function; see **Catching a Signal**, later.

Note that the **SIGKILL**, **SIGSTOP**, and **SIGABEND** signals can neither be caught nor ignored.

Table 11−8. Signals

| Name | Number | Default Action | Description |
|---|---|---|---|
| **SIGABEND** | 31 | Terminate with saveabend | Abnormal termination |
| **SIGABRT** | 6 | Terminate with saveabend | Abort process |
| **SIGALRM** | 14 | Terminate process | Alarm clock |
| **SIGCHLD** | 18 | Discard signal | Child has stopped or terminated |
| **SIGCONT** | 28 | Continue execution | Continue execution |
| **SIGFPE** | 8 | Terminate with saveabend | Arithmetic overflow |
| **SIGHUP** | 1 | Terminate process | Hangup |
| **SIGILL** | 4 | Terminate with saveabend | Invalid instruction |
| **SIGINT** | 2 | Terminate process | Interrupt |
| **SIGIO** | 7 | Discard signal | Input/output possible or completed |
| **SIGKILL** | 9 | Terminate process | Kill |
| **SIGLIMIT** | 27 | Terminate with saveabend | Operating system limits trap |
| **SIGMEMERR** | 22 | Terminate with saveabend | Uncorrectable memory error |
| **SIGMEMMGR** | 24 | Terminate with saveabend | Memory manager read error |
| **SIGNOMEM** | 23 | Terminate with saveabend | No memory available |
| **SIGPIPE** | 13 | Terminate process | Write on a pipe, no one to read it |
| **SIGQUIT** | 3 | Terminate with saveabend | Quit |
| **SIGRECV** | 19 | Discard signal | Message queued on $RECEIVE (currently not used) |
| **SIGSEGV** | 11 | Terminate with saveabend | Invalid address reference |
| **SIGSTK** | 25 | Terminate with saveabend | Stack overflow |
| **SIGSTOP** | 20 | Stop process | Stop |
| **SIGTERM** | 15 | Terminate process | Software termination signal |
| **SIGTIMEOUT** | 26 | Terminate process | Process loop timer timeout |
| **SIGTSTP** | 21 | Stop process | Interactive stop |
| **SIGTTIN** | 29 | Stop process | Background read attempted from controlling terminal |
| **SIGTTOU** | 30 | Stop process | Background write attempted to controlling terminal |
| **SIGUNCP** | 10 | Discard signal | Uncooperative process (H-series servers only) |
| **SIGURG** | 5 | Discard signal | Urgent condition on I/O channel |
| **SIGUSR1** | 16 | Terminate process | User-defined signal 1 |
| **SIGUSR2** | 17 | Terminate process | User-defined signal 2 |
| **SIGWINCH** | 12 | Discard signal | Terminal device window size changed |

For details of the process loop timer, see the SETLOOPTIMER procedure in the *Guardian Procedure Calls Reference Manual*.

**Note**: The process terminates and, where possible, a saveabend (core) file is created on generation of any signal if both of the following conditions are true:

- The signal was generated as a result of something other than the **kill( )** or **raise( )** function.
- Either the signal is blocked or ignored, or the process is in operating system code when the signal is generated.

The **_POSIX_JOB_CONTROL** symbolic constant is always defined. Hence, the job control signals are all supported: **SIGCHLD**, **SIGCONT**, **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, and **SIGTTOU**.

### Default Action

The default action for a signal occurs when the signal is not blocked and one of the following is true:

- No signal-catching function is installed for that signal.
- A signal-catching function is installed for the signal, but the **SIG_DFL** action is specified.

The table under **The Signals**, earlier, indicates a default action for each signal. The default actions have the following meanings:

Terminate process

Terminate the receiving process with all the consequences described for the **_exit( )** function.

Terminate with saveabend

Terminate the receiving process with all the consequences described for the **_exit( )** function. Create a memory image file (saveabend file) in the current directory of the receiving process if the following conditions are met:

- The effective user ID and the real user ID of the receiving process are equal.
- A regular file named **ZZSA***nnnn* (the saveabend file) can be created in the current directory. The file will have the following properties:
  - — The access permission code 0666 (0x1B6), modified by the filemode creation mask (see the **umask(2)** reference page)
  - — A file owner ID that is the same as the effective user ID of the receiving process
  - — A file group ID that is the same as the effective group ID of the receiving process

Note that the location of the saveabend file is different in the OSS environment than in the Guardian environment. In the Guardian environment, the file is created on the same subvolume as the program file. In the OSS environment, the file is created in the current working directory.

Saveabend files are known on most UNIX systems as core files.

Continue execution
: Restart the receiving process if it is stopped, or ignore the signal if the process is already executing.

Stop process
: Stop the execution of the receiving process. When a process stops, a **SIGCHLD** signal is sent to its parent process, unless the parent process has set the **SA_NOCLDSTOP** bit.

While a process is stopped, any additional signals that are sent to the process are not delivered until the process is continued. Exceptions to this are **SIGKILL** and **SIGABEND** signals, which always terminate the receiving process. Any other signal that causes process termination causes the same result as **SIGKILL** or **SIGABEND** except when they are blocked. The other exception is the **SIGCONT** signal, which causes the receiving process to restart or continue running, even if the signal is blocked or ignored.

Discard signal
: Ignore the signal. Delivery of the signal has no effect on the receiving process.

If a signal action is set to the **SIG_DFL** value while the signal is pending, the signal remains pending.

### Ignoring a Signal

A signal is ignored when a signal handler with the action set to **SIG_IGN** is installed for that signal.

Delivery of the signal has no effect on the receiving process.

Note that the **SIGKILL**, **SIGSTOP**, and **SIGABEND** signals cannot be ignored.

### Catching a Signal

Upon delivery of a signal that is to be caught, the receiving process is to run a signal-catching function specified in either the **sigaction( )** function call or the **signal( )** function call. The signal-catching function can be declared as follows:

**void handler(**
    **int** *signal***);**

The *signal* parameter is the signal number.

The process may choose to set up an alternate signal stack separate from the main process or thread stack to launch the signal handler. This option allows the catching of the **SIGSTK** signal, and any signals whose handler may overflow the process or thread stack. The signals to be caught on the alternate signal stack may be specified on a signal-by-signal basis. See **sigaction(2)** and **sigaltstack(2)** reference pages.

A new signal mask is calculated and installed for the duration of the signal-catching function or until a **sigprocmask( )** or **sigsuspend( )** function call is made. This mask is formed by taking the union of the process signal mask, the mask associated with the action for the signal being delivered, and a mask corresponding to the signal being delivered.

The mask associated with the signal-catching function is not allowed to block those signals that cannot be ignored. The system enforces this rule without causing an error to be indicated. If and when the signal-catching function returns, the original signal mask is restored and the receiving process resumes execution at the point it was interrupted.

The signal-catching function can cause the process to resume in a different context by calling the **siglongjmp( )** function. When the **siglongjmp( )** function is called, the process reverts to the state saved by a corresponding call to the **sigsetjmp( )** function.

Once the **sigaction( )** function installs an action for a specific signal, it remains installed until another action is explicitly requested by another call to the **sigaction( )** function or until a function from the **exec** or **tdm_exec** sets of functions is called. Signal actions installed by the **signal( )** function, however, are reset to the default action each time the signal is delivered.

If a signal action is set to a pointer to a function while the signal is pending, the signal remains pending.

When signal-catching functions are invoked asynchronously with process execution, the behavior of some of the functions is unspecified if they are called from a signal-catching function. The following is a list of functions that are reentrant. Therefore, applications can invoke these functions, without restriction, from signal-catching functions:

| | | | |
|---|---|---|---|
| **_exit( )** | **getegid( )** | **rmdir( )** | **tdm_spawnp( )** |
| **access( )** | **geteuid( )** | **setgid( )** | **tcdrain( )** |
| **alarm( )** | **getgid( )** | **setsid( )** | **tcflow( )** |
| **cfgetispeed( )** | **getgroups( )** | **setuid( )** | **tcflush( )** |
| **cfgetospeed( )** | **getpgrp( )** | **sigaction( )** | **tcgetattr( )** |
| **cfsetispeed( )** | **getpid( )** | **sigaddset( )** | **tcgetprgp( )** |
| **cfsetospeed( )** | **getppid( )** | **sigdelset( )** | **tcsendbreak( )** |
| **chdir( )** | **getuid( )** | **sigemptyset( )** | **tcsetattr( )** |
| **chmod( )** | **kill()** | **sigfillset( )** | **tcsetpgrp( )** |
| **chown( )** | **link( )** | **sigismember( )** | **time( )** |
| **close( )** | **lseek( )** | **signal( )** | **times( )** |
| **creat( )** | **mkdir( )** | **sigpending( )** | **umask( )** |
| **dup( )** | **mkfifo( )** | **sigprocmask( )** | **uname( )** |
| **dup2( )** | **open( )** | **sigsuspend( )** | **unlink( )** |
| **execle( )** | **pathconf( )** | **sleep( )** | **utime( )** |
| **execve( )** | **pause( )** | **stat( )** | **wait( )** |
| **fcntl( )** | **pipe( )** | **tdm_execvep( )** | **waitpid( )** |
| **fork( )** | **raise( )** | **tdm_execvp( )** | **write( )** |
| **fpathconf( )** | **read( )** | **tdm_fork( )** | |
| **fstat( )** | **rename( )** | **tdm_spawn( )** | |

All other functions are considered to be unsafe with respect to signals and should not be called from a signal-catching function, because their behavior is undefined.

On successful return from a signal-catching function for a **SIGFPE**, **SIGILL**, **SIGLIMIT**, **SIGMEMERR**, **SIGMEMMGR**, **SIGNOMEM**, **SIGSEGV**, or **SIGSTK** signal that was not generated by a **kill( )** or **raise( )** function call, a process receives a **SIGABEND** signal and terminates with a Guardian condition code of -11. A process deletion (-101) Guardian system message is sent to the mom, ancestor, or job ancestor of the terminated process and indicates abnormal termination.

No **SIGCHLD** signal is generated if a process establishes a signal-catching function for the **SIGCHLD** signal while the process has a terminated child process for which it has not waited.

## RELATED INFORMATION

Functions: **kill(2)**, **longjmp(3)**, **raise(3)**, **setjmp(3)**, **sigaction(2)**, **sigaddset(3)**, **sigaltstack(2)**, **sigdelset(3)**, **sigemptyset(3)**, **sigfillset(3)**, **sigismember(3)**, **siglongjmp(3)**, **signal(3)**, **sigpending(2)**, **sigprocmask(2)**, **sigsetjmp(3)**, **sigsuspend(2)**, **spt_sigwait(2)**.

## STANDARDS CONFORMANCE

The HP implementation does not provide the following signals defined in the IEEE Std 1003.1, 2004 Edition specification:

- **SIGBUS**, **SIGPOLL**, **SIGPROF**, **SIGSYS**, **SIGTRAP**, **SIGVTALRM**, **SIGXCPU**, and **SIGXFSZ**.

The POSIX standards leave some features to the implementing vendor to define. The following features are affected in the HP implementation:

- HP-specific signals are supported; see the extensions listed later.
- The **_POSIX_JOB_CONTROL** symbolic constant is always defined. Hence, the job control signals are all supported: **SIGCHLD**, **SIGCONT**, **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, and **SIGTTOU**.
- On generation of a blocked signal, it is unspecified whether the signal is discarded or remains pending when the associated action is to ignore the signal.
- Only one instance of the same signal can remain pending for a process. Subsequent occurrences of the same signal are discarded. However, this action is likely to change in a future release, so users should not depend on it.
- The order in which pending signals are delivered to a process is unspecified and should not be relied upon.
- Signals are generated for all operating system trap conditions normally detectable in the Guardian environment.
- After ignoring a **SIGFPE**, **SIGILL**, **SIGLIMIT**, **SIGMEMERR**, **SIGMEMMGR**, **SIGNOMEM**, **SIGSEGV**, or **SIGSTK** signal that was not generated by a **kill( )** or **raise( )** function, a process terminates. Similarly, after returning from a signal-catching function for one occurrence of such a signal, a process receives a **SIGABEND** signal and terminates with a Guardian condition code of -11. A process deletion (-101) Guardian system message is sent to the mom, ancestor, or job ancestor of the terminated process and indicates abnormal termination.
- No **SIGCHLD** signal is generated if a process establishes a signal-catching function for the **SIGCHLD** signal while the process has a terminated child process for which it has not waited.

The following are HP extensions to the IEEE Std 1003.1, 2004 Edition specification:

- The following signals are HP extensions: **SIGABEND**, **SIGIO**, **SIGLIMIT**, **SIGMEMERR**, **SIGMEMMGR**, **SIGNOMEM**, **SIGRECV**, **SIGSTK**, **SIGTIMEOUT**, **SIGWINCH**, and **SIGUNCP**.

## NAME

**spthread.h -** Thread-aware header file

## SYNOPSIS

**#include <spthread.h>**

## DESCRIPTION

The **<spthread.h>** header file contains the standard POSIX threads library API definitions. For reference, this reference page also documents the nonstandard POSIX extensions, thread-aware functions, thread-aware toolkit APIs, and thread-aware $RECEIVE APIs that are implemented in T1248 POSIX threads.

### Standard POSIX Threads Library APIs

The following are the standard POSIX threads library API definitions included in T1248 POSIX threads:

```
int pthread_atfork(
        pthread_void_fn_ptr_t, pthread_void_fn_ptr_t,
        pthread_void_fn_ptr_t );

int pthread_attr_destroy( pthread_attr_t * );

int pthread_attr_getdetachstate(
        const pthread_attr_t *, int * );

int pthread_attr_getinheritsched(
        const pthread_attr_t *, int * );

int pthread_attr_getschedparam(
        const pthread_attr_t *, struct sched_param * );

int pthread_attr_getschedpolicy(
        const pthread_attr_t *, int * );

int pthread_attr_getstackaddr(
        const pthread_attr_t *, void ** );

int pthread_attr_getstacksize(
        const pthread_attr_t *, size_t * );

int pthread_attr_init( pthread_attr_t * );

int pthread_attr_setdetachstate(
        pthread_attr_t *, int );

int pthread_attr_setinheritsched(
        pthread_attr_t *, int );

int pthread_attr_setschedparam(
        pthread_attr_t *,const struct sched_param * );

int pthread_attr_setschedpolicy(
        pthread_attr_t *, int );

int pthread_attr_setstacksize(
        pthread_attr_t *, size_t );

int pthread_cancel( pthread_t );

void pthread_cleanup_pop(
        int );
```

```
void pthread_cleanup_push(
        void  ( * ) ( void * ),
        void * );
int  pthread_cond_broadcast( pthread_cond_t * );
int  pthread_cond_destroy( pthread_cond_t * );
int  pthread_cond_init(
        pthread_cond_t *, const pthread_condattr_t * );
int  pthread_cond_signal( pthread_cond_t * );
int  pthread_cond_timedwait(
        pthread_cond_t *, pthread_mutex_t *,
        const struct timespec * );
int  pthread_cond_wait(
        pthread_cond_t *, pthread_mutex_t * );
int  pthread_condattr_destroy( pthread_condattr_t * );
int  pthread_condattr_init( pthread_condattr_t * );
int  pthread_create(
        pthread_t *, const pthread_attr_t *,
        pthread_startroutine_t,
        pthread_addr_t );
int  pthread_detach( pthread_t );
void  pthread_equal( thread1, thread2 );
void  pthread_exit( pthread_addr_t );
int  pthread_getschedparam(
        pthread_t, int *, struct sched_param * );
void * pthread_getspecific( pthread_key_t );
int  pthread_join( pthread_t, pthread_addr_t * );
int  pthread_key_create(
        pthread_key_t *, pthread_destructor_t );
int  pthread_key_delete( pthread_key_t );
int  pthread_kill( pthread_t, int );
int  pthread_mutexattr_destroy( pthread_mutexattr_t * );
int  pthread_mutexattr_init( pthread_mutexattr_t * );
int  pthread_mutex_destroy( pthread_mutex_t * );
int  pthread_mutex_init(
        pthread_mutex_t *, const pthread_mutexattr_t * );
int  pthread_mutex_lock( pthread_mutex_t * );
int  pthread_mutex_trylock( pthread_mutex_t * );
int  pthread_mutex_unlock( pthread_mutex_t * );
```

**int pthread_once(**
**pthread_once_t *, pthread_initroutine_t );**

**pthread_t pthread_self( void );**

**int pthread_setcancelstate( int, int * );**

**int pthread_setcanceltype( int, int * );**

**int pthread_setschedparam(**
**pthread_t, int, const struct sched_param * );**

**int pthread_setspecific(**
**pthread_key_t, pthread_addr_t );**

**int pthread_sigmask(**
**int, const sigset_t *, sigset_t * );**

**void pthread_testcancel( void );**

### Nonstandard POSIX API Definitions

The following are the nonstandard POSIX API definitions:

**int pthread_attr_getguardsize_np(**
**pthread_attr_t *, size_t * );**

**int pthread_attr_setguardsize_np(**
**pthread_attr_t *, size_t );**

**int pthread_cond_signal_int_np( pthread_cond_t * );**

**int pthread_delay_np( struct timespec *interval );**

**int pthread_getattr_np(**
**const pthread_t, pthread_attr_t ** );**

**int pthread_getconcurrency( void );**

**int pthread_getcontext_np(**
**pthread_t target, ucontext_t *ucp );**

**int pthread_get_expiration_np(**
**struct timespec *, struct timespec * );**

**int pthread_get_threadstateinfo_np(**
**pthread_t *tid,**
**char *state );**

**int pthread_kill_np(**
**pthread_t thread,**
**int sig );**

**int pthread_lock_global_np( void );**

**int pthread_mutexattr_getkind_np( pthread_mutexattr_t );**

**int pthread_mutexattr_setkind_np(**
**pthread_mutexattr_t *, int );**

**int pthread_setconcurrency( int new_level );**

**int pthread_signal_to_cancel_np(**
**sigset_t *, pthread_t * );**

```
int  pthread_unlock_global_np( void );

pid_t spt_fork( void );

int spt_getTMFConcurrentTransactions( void );

int spt_setTMFConcurrentTransactions( short );

short SPT_ABORTTRANSACTION( void );

short SPT_BEGINTRANSACTION( long * transaction_tag );

short SPT_BEGINTRANSACTION_EXT_ (
        int * transaction_tag,
        int timeout,
        long long * type_flags );

short SPT_ENDTRANSACTION( void );

short SPT_RESUMETRANSACTION( long transaction_tag );

short SPT_SERVERCLASS_DIALOG_BEGIN_ (
        long *dialogId,
        char * pathmon,
        short pathmonbytes,
        char * serverclass,
        short serverclassbytes,
        char * messagebuffer,
        short requestbytes,
        short maximumreplybytes,
        short * actualreplybytes,
        long timeout,
        short flags,
        short * scsoperationnumber,
        long tag );

short SPT_SERVERCLASS_DIALOG_BEGINL_ (
        long *dialogid,
        char * pathmon,
        short pathmonbytes,
        char * serverclass,
        short serverclassbytes,
        char * writebufferL,
        char * readbufferL,
        long requestbytes,
        long maximumreplybytes,
        long * actualreplybytes,
        long timeout,
        short flags,
        short * scsoperationnumber,
        long long tag );

short SPT_SERVERCLASS_DIALOG_SEND_ (
        long dialogId,
        char * messagebuffer,
        short requestbytes,
        short maximumreplybytes,
        short * actualreplybytes,
        long timeout,
        short flags,
```

```
            short * scsoperationnumber,
            long tag );
short SPT_SERVERCLASS_DIALOG_SENDL_ (
            long dialogid,
            char * writebufferL,
            char * readbufferL,
            long requestbytes,
            long maximumreplybytes,
            long * actualreplybytes,
            long timeout,
            short flags,
            short * scsoperationnumber,
            long long tag );
short SPT_SERVERCLASS_DIALOG_END_ ( long dialogId );
short SPT_SERVERCLASS_DIALOG_ABORT_ ( long dialogId );
short SPT_SERVERCLASS_SEND_INFO_ (
            short * serverclasserror,
            short * filesystemerror );
short SPT_SERVERCLASS_SEND_ (
            char * pathmon, short pathmonbytes,
            char * serverclass, short serverclassbytes,
            char * messagebuffer,
            short requestbytes, short maximumreplybytes,
            short * actualreplybytes,
            long timeout, short flags,
            short * scsoperationnumber, long tag );
short SPT_TMF_GetTxHandle(
            SPT_TMF_TxHandle_t *tx_handle );
short SPT_TMF_Init( void );
short SPT_TMF_RESUME( long long *txid );
short SPT_TMF_SetAndValidateTxHandle(
            SPT_TMF_TxHandle_t *tx_handle );
short SPT_TMF_SetTxHandle(
            SPT_TMF_TxHandle_t *tx_handle );
short SPT_TMF_SUSPEND( long long *txid );
spt_error_t spt_regPathsendFile( const short );
spt_error_t spt_regPathsendTagHandler(
            const long,
            spt_FileIOHandler_p,
            void * );
spt_error_t spt_unregPathsendTagHandler(
    const long );
int spt_sigaction(
    int, const struct sigaction *,
            struct sigaction * );
```

**unsigned int spt_sleep(**
**unsigned int );**

**int spt_usleep( unsigned int );**

**int spt_pause( );**

**int spt_sigpending( sigset_t * );**

**int spt_sigsuspend( const sigset_t * );**

**int spt_sigwait( sigset_t *, int * );**

### Thread-Aware Function Definitions

If invoking a thread-aware function using the corresponding native UNIX function, first define **SPT_THREAD_AWARE**. For example, after defining **SPT_THREAD_AWARE**, the function **printf** will resolve to the nonblocking function **spt_printf**.

The following are the nonstandard POSIX, HP-implemented thread-aware function definitions:

**int spt_accept(**
**int socket, struct sockaddr *address,**
**size_t *address_len );**

**int spt_close( int filedes );**

**int spt_connect(**
**int socket, const struct sockaddr *address,**
**size_t address_len );**

**int spt_fclose( FILE *stream );**

**int spt_fflush( FILE *stream );**

**int spt_fgetc( FILE *stream );**

**char *spt_fgets( char *string, int n, FILE *stream );**

**int spt_fgetwc( FILE *stream );**

**int spt_fprintf( FILE *stream, const char *format, ... );**

**int spt_fputc( int c, FILE *stream );**

**int spt_fputs( const char *string, FILE *stream );**

**wint_t spt_fputwc( wint_t c, FILE *stream );**

**ssize_t spt_fread(**
**void *pointer, size_t size,**
**size_t num_items, FILE *stream );**

**ssize_t spt_fwrite(**
**const void *pointer, size_t size,**
**size_t num_items, FILE *stream );**

**int spt_getc( FILE *stream );**

**int spt_getchar( void );**

**int spt_gets( FILE *stream );**

**int spt_getw( FILE *stream );**

```
wint_t spt_getwc( FILE *stream );
wint_t spt_getwchar( void );
int spt_printf( const char *format, ... );
int spt_putc( int c, FILE *stream );
int spt_putchar( int c );
int spt_puts( const char *string );
int spt_putw( int c, FILE *stream );
wint_t spt_putwc( wint_t c, FILE *stream );
wint_t spt_putwchar( wint_t c );
ssize_tvspt_readv(
        int filedes, struct iovec *iov, int iov_count );
ssize_t spt_recv(
        int socket, void *buffer,
        size_t length, int flags );
ssize_t spt_recvfrom(
        int socket, void *buffer,
        size_t length,int flags,
  struct sockaddr *address,
        size_t *address_len );
ssize_t spt_recvmsg(
        int socket, struct msghdr *message, int flags );
int spt_select(
        int nfds, fd_set *readfds,
        fd_set *writefds, fd_set *errorfds,
        struct timeval *timeout );
ssize_t spt_send(
        int socket, const void *buffer,
        size_t length, int flags );
ssize_t spt_sendmsg(
        int socket, const struct msghdr *message,
        int flags );
ssize_t spt_sendto(
        int socket, const void *buffer,
        size_t length, Lint flags,
        const struct sockaddr *dest_addr,
        size_t dest_len );
int spt_vfprintf(
        FILE *stream, const char *format,
        va_list printarg );
int spt_vprintf( const char *format, va_list printarg );
pid_t spt_waitpid(
        pid_t pid, int *stat_loc, int options );
```

**ssize_t spt_write(**
**int filedes, void *buffer, size_t nbytes );**

**ssize_t spt_writev(**
**int filedes, struct iovec *iov, int iov_count );**

### Thread-Aware Toolkit API Extensions

The following are the thread-aware toolkit API definitions:

**int spt_fd_read_ready(**
**const int fd, struct timeval *timeout );**

**int spt_fd_write_ready(**
**const int fd, struct timeval *timeout );**

**long spt_generateTag( void );**

**spt_error_t spt_interrupt(**
**const short filenum,**
**const spt_error_t errorSPT );**

**spt_error_t spt_interruptTag(**
**const short filenum, const long tag,**
**const, spt_error_t errorSPT );**

**spt_error_t spt_regFile( const short filenum );**

**spt_error_t spt_regFileIOHandler(**
**const short filenum,**
**const spt_FileIOHandler_p functionPtr );**

**spt_error_t spt_regOSSFileIOHandler( const int filedes,**
**const spt_OSSFileIOHandler_p functionPtr );**

**spt_error_t spt_regTimerHandler(**
**const spt_TimerHandler_p functionPtr );**

**spt_error_t spt_setOSSFileIOHandler(**
**const int filedes, const int read,**
**const int write,**
**const int error );**

**spt_error_t spt_unregFile( const short filenum );**

**spt_error_t spt_unregOSSFileIOHandler(**
**const int filedes );**

### Thread-Aware $RECEIVE API Definitions

The following are the thread-aware $RECEIVE API definitions:

**long spt_INITRECEIVE(**
**const short filenum,**
**const short receive_depth );**

**long spt_RECEIVEREAD( const short filenum,**
**char *buffer,**
**const short read_count, long *count_read,**
**const long timelimit, short *receive_info,**
**short *dialog_info );**

```
long spt_REPLYX(
        const char *buffer, const short write_count,
        short *count_written, const short msg_tag,
        const short error_return );
```

## NAME

**tar -** Describes the extended **tar** archive file format

## SYNOPSIS

**#include <tar.h>**

## DESCRIPTION

**tar** archives are created by the **tar** command. These archives are standardized and suitable for porting between different systems.

An extended **tar** archive or file consists of a series of blocks. Each block is a fixed size of 512 bytes.

Each file within the archive is represented by a header block and zero or more data blocks that contain the contents of the file. The header block describes the file. There are two blocks filled with binary zeros at the end of the archive as the end-of-archive indicator.

The data blocks are grouped for physical I/O. Each group of blocks is written with a single **write( )** operation. On magnetic tape, the result of this write is a single tape record.

The number of blocks in a group is set by the **-b** flag of the **tar** command; the default value is 20 blocks. The last group is always written at the full size, so blocks following the two zero blocks contain undefined data.

The header block is structured as shown in the following table. All lengths and offsets are in decimal.

Table 11–9. tar Archive File Header Block

| Field Name | Byte Offset | Length |
|---|---|---|
| **name** | 0 | |
| **mode** | 100 | 8 |
| **uid** | 108 | 8 |
| **gid** | 116 | 8 |
| **size** | 124 | 12 |
| **mtime** | 136 | 12 |
| **chksum** | 148 | 8 |
| **typeflag** | 156 | 1 |
| **linkname** | 157 | 100 |
| **magic** | 257 | 6 |
| **version** | 263 | 2 |
| **uname** | 265 | 32 |
| **gname** | 297 | 32 |
| **devmajor** | 329 | 8 |
| **devminor** | 337 | 8 |
| **prefix** | 345 | 155 |

Symbolic constants used in the header block are defined in the header file **/usr/include/tar.h**. The definitions are as follows:

```
#define TMAGIC          "ustar"         /* ustar and a null */
#define TMAGLEN         6
#define TVERSION        "00"            /* 00 and no null */
#define TVERSLEN        2

/* Values used in typeflag field */
#define REGTYPE         '0'             /* regular file */
#define AREGTYPE        ' '             /* regular file */
#define LNKTYPE         '1'             /* line         */
#define SYMTYPE         '2'             /* reserved     */
#define CHRTYPE         '3'             /* character special */
#define BLKTYPE         '4'             /* block special */
#define DIRTYPE         '5'             /* directory */
#define FIFOTYPE        '6'             /* FIFO special */
#define CONTTYPE        '7'             /* reserved */

/* Bits used in the mode field - values in octal */
#define TSUID           04000           /* set UID on execution */
#define TSGID           02000           /* set GID on execution */
#define TSVTX           01000           /* reserved             */
#define TUREAD          00400           /* read by owner        */
#define TUWRITE         00200           /* write by owner       */
#define TUEXEC          00100           /* execute/search by owner */
#define TGREAD          00040           /* read by group        */
#define TGWRITE         00020           /* write by group       */
#define TGEXEC          00010           /* execute/search by group */
#define TOREAD          00004           /* read by other        */
#define TOWRITE         00002           /* write by other       */
#define TOEXEC          00001           /* execute/search by other */
```

The fields in the header block must be contiguous; that is, no padding is used. Each character in the archive file is stored contiguously.

The fields **magic**, **uname**, and **gname** are null-terminated character strings.

The fields **name**, **linkname**, and **prefix** are null-terminated character strings except when all the characters in the field are nonnull characters, including the last character.

The **version** field is two bytes containing the characters "00" (ASCII zero-zero).

The **typeflag** field contains a single character.

All other fields are leading zero-filled octal numbers in ASCII. Each numeric field is terminated by one or more space or null characters.

The **name** and **prefix** fields produce the pathname of the file. The hierarchical relationship of the file is retained by specifying the pathname as a path prefix, a slash character, and the filename as the suffix. If **prefix** contains nonnull characters, then the value of **prefix**, a slash character, and the value of **name** are concatenated without modification or addition of new characters to produce a new pathname. In this manner, pathnames of at most 256 characters can be supported. If a pathname does not fit in the space provided, the **tar** command notifies the user of the error and no attempt is made to store any part of the file, header, or data in the archive.

The **linkname** field does not use **prefix** to produce a pathname. As such, **linkname** is limited to 99 characters. If the name does not fit in the space provided, the **tar** command notifies the user of the error and does not attempt to store the link in the archive.

The **mode** field contains 9 bits specifying file permissions and 3 bits specifying the set user ID (**TSUID**), set group ID (**TSGID**), and unused **TSVTX** modes. When the user restoring the files from the archive does not have appropriate permission to set these bits, the bits for which the user does not have permission are ignored.

The **uid** and **gid** fields are the user and group ID of the file's owner and group, respectively.

The **size** field is the size of the file in bytes, as follows:

- If the **typeflag** field specifies a file type of **LNKTYPE** or **SYMTYPE**, the **size** field is 0 (zero).
- If the **typeflag** field specifies a file type of **DIRTYPE**, the **size** field is interpreted as described for that record type.
- If the **typeflag** field specifies a file type of **CHARTYPE**, **BLKTYPE**, or **FIFOTYPE**, the meaning of the *size* field is implementation-defined and no data blocks are stored in the archive.
- If the **typeflag** specifies any other value, the number of blocks written following the header is (value of *size* + 511)/512, ignoring any fraction in the result of the division.

The **mtime** field is the modification time of the file at the time it was archived.

The **chksum** field is the ASCII representation of the octal value of the simple sum of all bytes in the header block. Each 8-bit byte in the header is treated as an unsigned value. These values are added to an unsigned integer that has been initialized to 0 (zero), and that has a precision of no less than 17 bits. When calculating the checksum, the **chksum** field is treated as if it were all blanks.

The **typeflag** field specifies the type of file archived. If a particular implementation does not recognize the type, or if the user does not have appropriate privilege to create that type, the file is extracted as if it were a regular file, if possible. If conversion to a regular file occurs, the **tar** command produces an error message indicating that the conversion took place.

The **magic** field indicates that the archive was output in **tar** archive file format. If this field contains the value **TMAGIC**, the **uname** and **gname** fields contain the ASCII representation of the owner and group of the file respectively. When the archive is restored, the password and group files are scanned for these names. If found, the user and group IDs from these files are used instead of the values contained within the **uid** and **gid** fields.

## RELATED INFORMATION

Commands: **pax(1)**, **pinstall(1)**.

## NAME

**termcap -** Describes the terminal capability database

## SYNOPSIS

**/etc/termcap**

## DESCRIPTION

The **termcap** database describes terminals. It is used, for example, by the **libtermcap** library. Terminals are described in **termcap** by giving a set of capabilities that they have and by describing how operations are performed. Padding requirements and initialization sequences are also included in **termcap**.

Entries in **termcap** consist of fields separated by colons (**:**). The first field for each terminal gives the names that are known for the terminal, separated by vertical bar (|) characters. The first name is always two characters long and is used by older systems that store the terminal name as a type in a 16-bit word in a system-wide database. The second name given is the most common abbreviation for the terminal. The last name given should be a long name fully identifying the terminal; all other names given are understood to be synonyms for the terminal name. All names but the first and last should be all lowercase letters and should contain no spaces; the last name can be in uppercase letters and can contain spaces for readability.

Terminal names (except for the last, long entry) should be chosen using the following conventions:

- The particular piece of hardware making up the terminal should have a root name chosen: for example, **hp2621**. This name should not contain dashes.
- Modes that the hardware can be in or user preferences should be indicated by appending a - (dash) and an indicator of the mode. Therefore, a DEC VT100 terminal in 132-column mode would be **vt100-w**.
- The suffixes in the following table should be used where possible.

Table 11−10. Terminal Name Suffixes

| Suffix | Meaning | Example |
|---|---|---|
| **-w** | Wide mode (more than 80 columns) | **vt100-w** |
| **-am** | With automatic margins (usually default) | **vt100-am** |
| **-nam** | Without automatic margins | **vt100-nam** |
| **-***n* | Number of lines on the screen | **aaa-60** |
| **-na** | No arrow keys (leave them in local) | **concept100-na** |
| **-***n***p** | Number of pages of memory | **concept100-4p** |
| **-rv** | Reverse video | **concept100-rv** |

### Terminal Types Supported

When using TELNET to connect to Open System Services, only the terminal type **vt100** is supported. Other terminal types described in this reference page are not supported.

### Types of Capabilities

All capabilities have 2-letter name codes, such as **cm**.

Capabilities in **termcap** are of three types:

bool Boolean capabilities, which indicate particular features that the terminal has.

num Numeric capabilities, which give the size of the display or the size of other attributes.

str String capabilities, which give character sequences that can be used to perform particular terminal operations.

The following table describes the capabilities used to describe terminals.

**Notes to the table:**

N Indicates numeric parameter(s).

P Indicates that padding can be specified.

* Indicates that padding can be based on the number of lines affected.

o Indicates that the capability is obsolete. New software should not rely on this capability at all.

Table 11−11. Terminal Capabilities

| Name | Type | Notes | Description |
|---|---|---|---|
| **ae** | str | (P) | End alternate character set |
| **AL** | str | (NP*) | Add *n* new blank lines |
| **al** | str | (P*) | Add new blank line |
| **am** | bool | | Terminal has automatic margins |
| **as** | str | (P) | Start alternate character set |
| **bc** | str | (o) | Backspace if not **ˆH** |
| **bl** | str | (P) | Audible signal (bell) |
| **bs** | bool | (o) | Terminal can backspace with **ˆH** |
| **bt** | str | (P) | Back tab |
| **bw** | bool | | The **le** (backspace) wraps from column 0 to last column |
| **CC** | str | | Terminal settable command character in prototype |
| **cd** | str | (P*) | Clear to end of display |
| **ce** | str | (P) | Clear to end of line |
| **ch** | str | (NP) | Set cursor column (horizontal position) |
| **cl** | str | (P*) | Clear screen and home cursor |
| **CM** | str | (NP) | Memory-relative cursor addressing |
| **cm** | str | (NP) | Screen-relative cursor motion |
| **co** | num | | Number of columns in a line (see **NOTES** section) |
| **cr** | str | (P) | Carriage return |
| **cs** | str | (NP) | Change scrolling region (DEC VT100 terminal) |
| **ct** | str | (P) | Clear all tab stops |
| **cv** | str | (NP) | Set cursor row (vertical position) |
| **da** | bool | | Display can be retained above the screen |
| **dB** | num | (o) | Milliseconds of **bs** delay needed (default 0) |
| **db** | bool | | Display can be retained below the screen |
| **DC** | str | (NP*) | Delete *n* characters |
| **dC** | num | (o) | Milliseconds of **cr** delay needed (default 0) |
| **dc** | str | (P*) | Delete character |
| **dF** | num | (o) | Milliseconds of **ff** delay needed (default 0) |
| **DL** | str | (NP*) | Delete *n* lines |
| **dl** | str | (P*) | Delete line |

| | | | |
|---|---|---|---|
| **dm** | str | | Enter delete mode |
| **dN** | num | (o) | Milliseconds of **nl** delay needed (default 0) |
| **DO** | str | (NP*) | Move cursor down *n* lines |
| **do** | str | | Down one line |
| **ds** | str | | Disable status line |
| **dT** | num | (o) | Milliseconds of horizontal tab delay needed (default 0) |
| **dV** | num | (o) | Milliseconds of vertical tab delay needed (default 0) |
| **ec** | str | (NP) | Erase *n* characters |
| **ed** | str | | End delete mode |
| **ei** | str | | End insert mode |
| **eo** | bool | | Can erase overstrikes with a blank |
| **EP** | bool | (o) | Even parity |
| **es** | bool | | Escape can be used on the status line |
| **ff** | str | (P*) | Hard-copy terminal page eject |
| **fs** | str | | Return from status line |
| **gn** | bool | | Generic line type (for example, dialup, switch) |
| **hc** | bool | | Hardcopy terminal |
| **HD** | bool | (o) | Half-duplex |
| **hd** | str | | Half-line down (forward 1/2 linefeed) |
| **ho** | str | (P) | Home cursor |
| **hs** | bool | | Has extra status line |
| **hu** | str | | Half-line up (reverse 1/2 linefeed) |
| **hz** | bool | | Cannot print **˜s** (Hazeltine) |
| **i1-i3** | str | | Terminal initialization strings (unsupported) |
| **IC** | str | (NP*) | Insert *n* blank characters |
| **ic** | str | (P*) | Insert character |
| **if** | str | | Name of file containing initialization string |
| **im** | str | | Enter insert mode |
| **in** | bool | | Insert mode distinguishes nulls |
| **iP** | str | | Pathname of program for initialization (unsupported) |
| **ip** | str | (P*) | Insert pad after character inserted |
| **is** | str | | Terminal initialization string (**termcap** only) |
| **it** | num | | Tabs initially every *n* positions |
| **K1** | str | | Sent by keypad upper left |
| **K2** | str | | Sent by keypad upper right |
| **K3** | str | | Sent by keypad center |
| **K4** | str | | Sent by keypad lower left |
| **K5** | str | | Sent by keypad lower right |
| **k0-k9** | str | | Sent by function keys 0 to 9 |
| **kA** | str | | Sent by insert-line key |
| **ka** | str | | Sent by clear-all-tabs key |
| **kb** | str | | Sent by backspace key |
| **kC** | str | | Sent by clear-screen or erase key |
| **kD** | str | | Sent by delete-character key |
| **kd** | str | | Sent by down-arrow key |
| **kE** | str | | Sent by clear-to-end-of-line key |
| **ke** | str | | Out of keypad transmit mode |
| **kF** | str | | Sent by scroll-forward/down key |

| | | | |
|---|---|---|---|
| **kH** | str | | Sent by home-down key |
| **kh** | str | | Sent by home key |
| **kI** | str | | Sent by insert-character or enter-insert-mode key |
| **kL** | str | | Sent by delete-line key |
| **kl** | str | | Sent by left-arrow key |
| **kM** | str | | Sent by insert key while in insert mode |
| **km** | bool | | Has a meta key (shift, sets parity bit) |
| **kN** | str | | Sent by next-page key |
| **kn** | num | (o) | Number of function (**k0-k9**) keys (default 0) |
| **ko** | str | (o) | The **termcap** entries for other nonfunction keys |
| **kP** | str | | Sent by previous-page key |
| **kR** | str | | Sent by scroll-backward/up key |
| **kr** | str | | Sent by right-arrow key |
| **kS** | str | | Sent by clear-to-end-of-screen key |
| **ks** | str | | Put terminal in keypad transmit mode |
| **kT** | str | | Sent by set-tab key |
| **kt** | str | | Sent by clear-tab key |
| **ku** | str | | Sent by up-arrow key |
| **l0-l9** | str | | Labels on function keys if not "**f***n*" |
| **LC** | bool | (o) | Lowercase characters only |
| **LE** | str | (NP) | Move cursor left *n* positions |
| **le** | str | (P) | Move cursor left one position |
| **li** | num | | Number of lines on screen or page (see **NOTES** section) |
| **ll** | str | | Last line, first column |
| **lm** | num | | Lines of memory if > **li** (0 means varies) |
| **ma** | str | (o) | Arrow key map (used by **vi** version 2 only) |
| **mb** | str | | Turn on blinking attribute |
| **md** | str | | Turn on bold (extra bright) attribute |
| **me** | str | | Turn off all attributes |
| **mh** | str | | Turn on half-bright attribute |
| **mi** | bool | | Safe to move while in insert mode |
| **mk** | str | | Turn on blank attribute (characters invisible) |
| **ml** | str | (o) | Memory lock on above cursor |
| **mm** | str | | Turn on meta mode (eighth bit) |
| **mo** | str | | Turn off meta mode |
| **mp** | str | | Turn on protected attribute |
| **mr** | str | | Turn on reverse-video attribute |
| **ms** | bool | | Safe to move in standout modes |
| **mu** | str | (o) | Memory unlock (turn off memory lock) |
| **nc** | bool | (o) | No correctly working **cr** (Datamedia 2500, Hazeltine 2000 terminals) |
| **nd** | str | | Nondestructive space (cursor right) |
| **NL** | bool | (o) | The **\n** is newline, not linefeed |
| **nl** | str | (o) | Newline character if not **\n** |
| **ns** | bool | (o) | Terminal is a CRT, but does not scroll |
| **nw** | str | (P) | Newline (behaves like **cr** followed by **do**) |
| **OP** | bool | (o) | Odd parity |
| **os** | bool | | Terminal overstrikes |

| | | | |
|---|---|---|---|
| **pb** | num | | Lowest baud where delays are required |
| **pc** | str | | Pad character (default NULL) |
| **pf** | str | | Turn off the printer |
| **pk** | str | | Program function key *n* to type string *s* (unsupported) |
| **pl** | str | | Program function key *n* to execute string *s* (unsupported) |
| **pO** | str | (N) | Turn on the printer for *n* bytes |
| **po** | str | | Turn on the printer |
| **ps** | str | | Print contents of the screen |
| **pt** | bool | (o) | Has hardware tabs (might need to be set with **is**) |
| **px** | str | | Program function key *n* to transmit string *s* (unsupported) |
| **r1-r3** | str | | Reset terminal completely to sane modes (unsupported) |
| **rc** | str | (P) | Restore cursor to position of last **sc** |
| **rf** | str | | Name of file containing reset codes |
| **RI** | str | (NP) | Move cursor right *n* positions |
| **rp** | str | (NP*) | Repeat character *c n* times |
| **rs** | str | | Reset terminal completely to sane modes (**termcap** only) |
| **sa** | str | (NP) | Define the video attributes |
| **sc** | str | (P) | Save cursor position |
| **se** | str | | End standout mode |
| **SF** | str | (NP*) | Scroll forward *n* lines |
| **sf** | str | (P) | Scroll text up |
| **sg** | num | | Number of garbage characters left by **so** or **se** (default 0) |
| **so** | str | | Begin standout mode |
| **SR** | str | (NP*) | Scroll backward *n* lines |
| **sr** | str | (P) | Scroll text down |
| **st** | str | | Set a tab in all rows, current column |
| **ta** | str | (P) | Tab to next 8-position hardware tab stop |
| **tc** | str | | Entry of similar terminal (must be last) |
| **te** | str | | String to end programs that use **termcap** |
| **ti** | str | | String to begin programs that use **termcap** |
| **ts** | str | (N) | Go to status line, column *n* |
| **UC** | bool | (o) | Uppercase characters only |
| **uc** | str | | Underscore one character and move past it |
| **ue** | str | | End underscore mode |
| **ug** | num | | Number of garbage characters left by **us** or **ue** (default 0) |
| **ul** | bool | | Underline character overstrikes |
| **UP** | str | (NP*) | Move cursor up *n* lines |
| **up** | str | | Up a line (cursor up) |
| **us** | str | | Start underscore mode |
| **vb** | str | | Visible bell (must not move cursor) |
| **ve** | str | | Make cursor appear normal (undo **vs**/**vi**) |
| **vi** | str | | Make cursor invisible |
| **vs** | str | | Make cursor very visible |
| **vt** | num | | Virtual terminal number (not supported on all systems) |
| **wi** | str | (N) | Set current window |
| **ws** | num | | Number of columns in status line |
| **xb** | bool | | Beehive (**f1=ESC**, **f2=ˆC**) |

| | | | |
|---|---|---|---|
| **xn** | bool | | Newline ignored after 80 columns (Concept) |
| **xo** | bool | | Terminal uses **xoff**/**xon** (DC3/2DC1) handshaking |
| **xr** | bool | (o) | Return acts like **ce cr nl** (Delta Data) |
| **xs** | bool | | Standout not erased by overwriting (Hewlett-Packard terminals) |
| **xt** | bool | | Tabs ruin magic **so** character (Teleray 1061 terminal) |
| **xx** | bool | (o) | Tektronix 4025 terminal insert-line |

### A Sample Entry

The following entry, which describes the Concept-100, is among the more complex entries in the **termcap** file as of this writing:

```
ca | concept100 | c100 | concept | c104 | concept100-4p | HDS Concept-100:\
    :al=3*\E^R:am:bl=^G:cd=16*\E^C:ce=16\E^U:cl=2*^L:cm=\Ea%+ %+ :\
    :co#80:.cr=9^M:db:dc=16\E^A:dl=3*\E^B:do=^J:ei=\E\200:eo:im=\E^P:in:\
    :ip=16*:is=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200\Eo\47\E:k1=\E5:\
    :k2=\E6:k3=\E7:kb=^h:kd=\E<:ke=\Ex:kh=\E?:kl=\E>:kr=\E=:ks=\EX:\
    :ku=\E;:le=^H:li#24:mb=\EC:me=\EN\200:mh=\EE:mi:mk=\EH:mp=\EI:\
    :mr=\ED:nd=\E=:pb#9600:rp=0.2*\Er%.%+ :se=\Ed\Ee:sf=^J:so=\EE\ED:\
    :.ta=8\t:te=\Ev    \200\200\200\200\200\200\Ep\r\n:\
    :ti=\EU\Ev  8p\Ep\r:ue=\Eg:ul:up=\E;:us=\EG:\
    :vb=\Ek\200\200\200\200\200\200\200\200\200\200\200\200\200\200\EK:\
    :ve=\Ew:vs=\EW:vt#8:xn:\
    :bs:cr=^M:dC#9:dT#8:nl=^J:ta=^I:pt:
```

Entries can continue onto multiple lines by using a \ (backslash) as the last character of a line, and empty fields can be included for readability (here between the last field on a line and the first field on the next). Comments can be included on lines beginning with a **#** (number sign) character.

For instance, the fact that the Concept terminal has automatic margins (that is, an automatic return and linefeed when the end of a line is reached) is indicated by the Boolean capability **am**. Hence the description of the Concept includes **am**.

In the **termcap** file, numeric capabilities are followed by the **#** (number sign) character and the value. In the preceding example, the **co** capability, which indicates the number of columns in the display, has the value 80 for the Concept terminal.

In the **termcap** file, string-valued capabilities such as **ce** (clear-to-end-of-line sequence) are given by the two-letter capability code, an **=** (equal sign), then a string ending at the next following **:** (colon).

A delay in milliseconds can appear after the **=** in such a capability, which causes padding characters to be supplied after the remainder of the string is sent to provide this delay. The delay can be either a number (for example, 20), or a number followed by an ***** (for example, 3*). An ***** (asterisk) indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-line padding required. (In the case of insert-character, the factor is still the number of lines affected; this is always a 1 unless the terminal has **in** capability and the software uses it.) When an ***** (asterisk) is specified, it is sometimes useful to give a delay of the form 3.5 to specify a delay per line to tenths of milliseconds. (Only one decimal place is allowed.)

A number of escape sequences are provided in the string-valued capabilities for easy encoding of control characters there. **\E** maps to an ESC character, *ˆX* maps to a <**Ctrl**-*X*> for any appropriate *X*, and the sequences **\n**, **\r**, **\t**, **\b**, and **\f** map to linefeed, return, tab, backspace, and formfeed, respectively. Finally, characters can be given as three octal digits after a \ (backslash), and the characters **ˆ** (circumflex) and \ can be given as \ˆ and \\. If it is necessary to place a **:** (colon) in a capability, it must be encoded in octal as **\072**. If it is necessary to place a NUL character in a string capability, it must be encoded as **\200**. (The routines that deal with **termcap** use C strings and strip the high bits off the output very late, so that **\200** has the same result as **\000**.)

In the **termcap** file, individual capabilities must sometimes be commented out. To do this, put a **.** (dot) before the capability name. For example, see the first **cr** and **ta** capabilities in the preceding example.

### Basic Capabilities

The number of columns on each line of the terminal display is given by the **co** numeric capability. If the display is a CRT, then the number of lines on the screen is given by the **li** capability. If the display wraps around to the beginning of the next line when the cursor reaches the right margin, then indicate this with the **am** capability. If the terminal can clear its screen, indicate this with the **cl** string capability. If the terminal overstrikes (rather than clearing the position when a character is overwritten), indicate this with the **os** capability. If the terminal is a printing terminal, with no soft copy unit, indicate this with both the **hc** and **os** capabilities. (**os** applies to storage scope terminals, such as the Tektronix 4010 series, as well as to hard-copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, indicate this with the **cr** capability. (Normally this is carriage return, **^M**.) If there is a code to produce an audible signal (bell, beep, and so forth), indicate this with the **bl** capability.

If there is a code (such as a backspace) to move the cursor one position to the left, indicate this with the **le** capability. Similarly, codes to move to the right, up, and down should be indicated with the **nd**, **up**, and **do** capabilities, respectively. These local cursor motions should not alter the text they pass over; for example, you would not normally use **nd=** unless the terminal has the **os** capability, because the space would erase the character moved over.

Note that the local cursor motions encoded in **termcap** have undefined behavior at the left and top edges of a CRT display. Applications should never attempt to backspace around the left edge, unless **bw** is given, and never attempt to go up off the top using local cursor motions.

To scroll text up, an application goes to the bottom left corner of the screen and sends the string given by the **sf** (index) capability. To scroll text down, an application goes to the top left corner of the screen and sends the string given by the **sr** (reverse index) capability. The strings given by **sf** and **sr** have undefined behavior when not on their respective corners of the screen. Parameterized versions of the scrolling sequences are the **SF** and **SR** capabilities, which have the same semantics as **sf** and **sr** except that they take one parameter and scroll that many lines. They also have undefined behavior except at the appropriate corner of the screen.

The **am** capability indicates whether the cursor sticks at the right edge of the screen when text is output there, but this action does not necessarily apply to the action of the **nd** capability from the last column. Leftward local motion is defined from the left edge only when the **bw** capability is given; then the action of the **le** capability from the left edge moves to the right edge of the previous row. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch-selectable automatic margins, the **termcap** description usually assumes that this feature is on (that is, assumes the **am** capability). If the terminal has a command that moves to the first column of the next line, that command can be given as the **nw** (newline) capability. It is permissible for this action to clear the remainder of the current line, so if the terminal has no correctly working CR and LF, it is still possible to create a working **nw** capability out of one or both of them.

These capabilities suffice to describe hard-copy and "glass-tty" terminals. Thus, the Teletype model 33 is described as follows:

```
T3|tty33|33|tty|Teletype model 33:\
        :bl=^G:co#72:cr=^M:do=^J:hc:os:
```

and the Lear Siegler ADM-3 terminal is described as follows:

```
l3|adm3|3|LSI ADM-3:\
```

**:am:bl=ˆG:cl=ˆZ:co#80:cr=ˆM:do=ˆJ:le=ˆH:li#24:sf=ˆJ:**

### Parameterized Strings

Cursor addressing and other strings requiring parameters are described by a parameterized string capability, with escape encodings like those of the **printf( )** function (**%***x*) in it, while other characters are passed through unchanged. For example, to address the cursor, the **cm** capability is given, using two parameters: the row and column to move to. Rows and columns are numbered from 0 (zero) and refer to the physical screen visible to the user, not to any unseen memory. If the terminal has memory-relative cursor addressing, that can be indicated by the analogous **CM** capability.

The **%** encodings have the following meanings:

| | |
|---|---|
| **%%** | Writes **%** |
| **%d** | Writes value as **printf( ) %d** encoding does |
| **%2** | Writes value as **printf( ) %2d** encoding does |
| **%3** | Writes value as **printf( ) %3d** encoding does |
| **%.** | Writes value as **printf( ) %c** encoding does |
| **%+***x* | Adds *x* to value, then writes **%** |
| **%>***xy* | If value > *x* then adds *y* (no output) |
| **%r** | Reverses order of two parameters (no output) |
| **%i** | Increments by 1 (one) (no output) |
| **%n** | Exclusive ORs all parameters with 0140 (Datamedia 2500) |
| **%B** | BCD (16*(value/10)) + (value%10) (no output) |
| **%D** | Reverse coding (value - 2*(value%16)) (no output) (Delta Data terminal) |

Consider the Hewlett-Packard 2645 terminal, which, to get to row 3 and column 12, needs to be sent **\E&a12c03Y** padded for 6 milliseconds. Note that the order of the row and column coordinates is reversed here and that the row and column are sent as 2-digit integers. Thus, its **cm** capability is **cm=6\E&%r%2c%2Y**.

The Microterm ACT-IV terminal needs the current row and column sent simply encoded in binary preceded by a **ˆT**, **cm=ˆT%.%.**. Terminals that use **%.** need to be able to backspace the cursor (the **le** capability) and to move the cursor up one line on the screen (the **up** capability). This is necessary because it is not always safe to transmit **\n**, **ˆD**, and **\r**, as the system can change or discard them. (Applications using **termcap** must set terminal modes so that tabs are not expanded, so **\t** is safe to send. This is essential for the Ann Arbor 4080 terminal.)

A final example is the Lear Siegler ADM-3a terminal, which offsets row and column by a blank character; thus, **cm=\E=%+ %+**.

Row or column absolute cursor addressing can be given as single-parameter capabilities **ch** (horizontal position absolute) and **cv** (vertical position absolute). Sometimes these are shorter than the more general two-parameter sequence (as with the Hewlett-Packard 2645 terminal) and can be used in preference to **cm**. If there are parameterized local motions (for example, move *n* positions to the right), these can be given as the **DO**, **LE**, **RI**, and **UP** capabilities with a single parameter indicating how many positions to move. These capabilities are primarily useful for terminals that do not have the **cm** capability, such as the Tektronix 4025 terminal.

### Cursor Motions

If the terminal has a fast way to home the cursor (to the upper left-hand corner of the screen), this can be given as the **ho** capability. Similarly, a fast way of getting to the lower left-hand corner can be given as the **ll** capability; this way can involve going up with the **up** capability from the home position, but an application should never do this itself (unless **ll** does), because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as cursor address (0,0): to the top left corner of the screen, not of memory. Therefore, the **\EH** sequence on Hewlett-Packard terminals cannot be used for the **ho** capability.

### Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as the **ce** capability. If the terminal can clear from the current position to the end of the display, this should be given as the **cd** capability. **cd** must be invoked only from the first column of a line. (Therefore, it can be simulated by a request to delete a large number of lines, if a true **cd** capability is not available.)

### Insert/Delete Line

If the terminal can open a new blank line before the line containing the cursor, this should be given as the **al** capability; this action must be invoked only from the first position of a line. The cursor must then appear at the left of the newly blank line. If the terminal can delete the line that the cursor is on, this should be given as the **dl** capability; this action must be used only from the first position on the line to be deleted. Versions of **al** and **dl** that take one parameter and insert or delete the indicated number of lines can be given as the **AL** and **DL** capabilities.

If the terminal has a settable scrolling region (like the DEC VT100 terminal), the command to set this can be described with the **cs** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, therefore, undefined after using this capability. It is possible to get the effect of insert or delete line using this capability — the **sc** and **rc** (save and restore cursor) capabilities are also useful. Inserting lines at the top or bottom of the screen can also be done using the **sr** or **sf** capability on many terminals without a true insert/delete line feature, and **sr** and **sf** are often faster even on terminals with insert/delete line features.

Terminals with the "magic cookie" glitches (the **sg** and **ug** capabilities), rather than maintaining extra attribute bits for each character cell, instead deposit special cookies, or garbage characters, when they receive mode-setting sequences, which affect the display algorithm.

Some terminals, such as the Hewlett-Packard 2621 terminal, automatically leave standout mode when they move to a newline or when the cursor is addressed. Applications using standout mode should exit standout mode on such terminals before moving the cursor or sending a newline. On terminals where this is not a problem, the **ms** capability should be present to indicate that this overhead is unnecessary.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement), this can be indicated by the **vb** capability; this action must not move the cursor.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to change, for example, a nonblinking underline into an easier-to-find block or blinking underline), indicate the **vs** capability. If there is a way to make the cursor completely invisible, indicate the **vi** capability. The **ve** capability, which undoes the effects of both of these modes, should also be specified.

If your terminal correctly displays underlined characters (with no special codes needed), even though it does not overstrike, then you should specify the **ul** capability. If overstrikes can be erased with a blank, indicate that by specifying the **eo** capability.

## Keypad Support

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be provided in **termcap**. Note that it is not possible to handle terminals where the keypad works only in local mode (this applies, for example, to the unshifted Hewlett-Packard 2621 terminal's keys). If the keypad can be set to transmit or not transmit, specify the **ks** and **ke** capabilities. Otherwise, the keypad is assumed to always transmit. The codes sent by the left-arrow, right-arrow, up-arrow, down-arrow, and home keys can be specified by the **kl**, **kr**, **ku**, **kd**, and **kh** capabilities, respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be specified by the **k0**, **k1**,..., **k9** capabilities. If these keys have labels other than the default f0 through f9, the labels can be specified by the **l0**, **l1**,..., **l9** capabilities.

The codes transmitted by certain other special keys can be specified by the following capabilities:

| | |
|---|---|
| **kH** | Home down |
| **kb** | Backspace |
| **ka** | Clear all tabs |
| **kt** | Clear the tab stop in this column |
| **kC** | Clear screen or erase |
| **kD** | Delete character |
| **kL** | Delete line |
| **kM** | Exit insert mode |
| **kE** | Clear to end of line |
| **kS** | Clear to end of screen |
| **kI** | Insert character or enter insert mode |
| **kA** | Insert line |
| **kN** | Next page |
| **kP** | Previous page |
| **kF** | Scroll forward/down |
| **kR** | Scroll backward/up |
| **kT** | Set a tab stop in this column |

In addition, if the keypad has a 3-by-3 array of keys including the four arrow keys, then the other five keys can be specified by the **K1**, **K2**, **K3**, **K4**, and **K5** capabilities. These keys are useful when the effects of a 3-by-3 directional pad are needed. The obsolete **ko** capability, used to describe "other" function keys, has been completely replaced by the preceding list of capabilities.

The **ma** capability is also used to indicate arrow keys on terminals that have single-character arrow keys. It is obsolete but still in use in version 2 of the **vi** utility, which must be run on some minicomputers due to memory limitations. The **ma** capability is redundant with the **kl**, **kr**, **ku**, **kd**, and **kh** capabilities. The **ma** capability consists of groups of two characters. In each group, the first character is what an arrow key sends, and the second character is the corresponding **vi** command. These **vi** commands are **h** for the **kl** capability, **j** for **kd**, **k** for **ku**, **l** for **kr**, and **H** for **kh**. For example, the Microterm Mime terminal would have **ma=^Hh^Kj^Zk^Xl** indicating arrow keys left (**^H**), down (**^K**), up (**^Z**), and right (**^X**). (There is no home key on the Microterm Mime terminal.)

### Tabs and Initialization

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be specified by the **ti** and **te** capabilities. This need arises, for example, from terminals like the Concept terminal with more than one page of memory. If the terminal has only memory-relative cursor addressing and not screen-relative cursor addressing, a screen-sized window must be fixed into the display for cursor addressing to work properly. This is also used for the Tektronix 4025 terminal, where the **ti** capability sets the command character to be the one used by **termcap**.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the **termcap** description. They are printed in the following order:

- **is**
- setting tabs using the **ct** and **st** capabilities
- **if**

A pair of sequences that does a harder reset from a totally unknown state can be analogously given as the **rs** and **if** capabilities. These strings are output by the **reset** program, which is used when the terminal gets into a wedged state. Commands are normally placed in the **rs** and **rf** capabilities only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set the DEC VT100 terminal into 80-column mode would normally be part of the **is** capability, but it causes annoying behavior of the screen and is not normally needed because the terminal is usually already in 80-column mode.

If the terminal has hardware tabs, the command to advance to the next tab stop can be indicated by the **ta** capability (usually **ˆI**). A backtab command that moves leftward to the previous tab stop can be indicated by the **bt** capability. By convention, if the terminal driver modes indicate that tab stops are being expanded by the computer rather than being sent to the terminal, applications should not use **ta** or **bt**, even if they are present, because the user might not have the tab stops properly set. If the terminal has hardware tabs that are initially set every *n* positions when the terminal is powered up, then the numeric **it** capability is given, indicating the number of positions between tab stops.

If there are commands to set and clear tab stops, they can be given as the **ct** (clear all tab stops) capability and the **st** (set a tab stop in the current column of every row) capability. If a more complex sequence is needed to set the tabs than can be described by this, the sequence can be placed in the **is** or **if** capabilities.

### Delays

Certain capabilities control padding in the terminal driver. Delays embedded in the capabilities **cr**, **sf**, **le**, **ff**, and **ta** cause the appropriate delay bits to be set in the terminal driver. If the **pb** capability (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

### Miscellaneous

If the terminal requires other than a NULL (zero) character as a pad, this can be indicated by the **pc** capability. Only the first character of the **pc** string is used.

If the terminal has commands to save and restore the position of the cursor, indicate this with the **sc** and **rc** capabilities.

If the terminal has an extra status line that is not normally used by software, this fact can be indicated in **termcap**. If the status line is viewed as an extra line below the bottom line, then the **hs** capability should be given.

Special strings to go to a position in the status line and to return from the status line can be given as the **ts** and **fs** capabilities. (Note that **fs** must leave the cursor position in the same place that it was before **ts**. If necessary, the strings from the **sc** and **rc** capabilities can be included in **ts** and **fs** to get this effect.) The **ts** capability takes one parameter, which is the column number of the status line to which the cursor is to be moved.

If escape sequences and other special commands such as tab work while in the status line, the **es** capability can be given. A string that turns off the status line (or otherwise erases its contents) should be indicated by the **ds** capability.

The status line is normally assumed to be the same width as the rest of the screen; that is, the value indicated in the **co** capability. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded), then its width in columns can be indicated with the numeric **ws** capability.

If the terminal can move up or down half a line, this can be indicated with the **hu** (half-line up) and **hd** (half-line down) capabilities. This feature is primarily useful for superscripts and subscripts on hard-copy terminals. If a hard-copy terminal can eject to the next page (formfeed), indicate this with the **ff** capability (usually **^L**).

If the terminal has a settable command character, such as the Tektronix 4025 terminal, indicate this with the **CC** capability. A prototype command character is chosen that is used in all capabilities. This character is given in the **CC** capability to identify it. The following convention is supported on some UNIX systems: the environment is searched for a **CC** variable, and if found, all occurrences of the prototype character are replaced by the character in the environment variable. Do not use the **CC** environment variable in this way; it conflicts with the **make** command.

Terminal descriptions that do not represent a specific kind of known terminal, such as **switch**, **dialup**, **patch**, and **network**, should include the **gn** (generic) capability so that applications can explain that they do not know how to talk to the terminal. (This capability does not apply to **virtual** terminal descriptions for which the escape sequences are known.)

If the terminal uses **xoff**/**xon** (DC3/DC1) handshaking for flow control, indicate this with the **xo** capability. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters are not transmitted.

If the terminal has a meta key that acts as a shift key by setting the 8th bit of any character transmitted, then this fact can be indicated with the **km** capability. Otherwise, software assumes that the 8th bit is parity and it will usually be cleared. If strings exist to turn this meta mode on and off, they can be given by the **mm** and **mo** capabilities.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with the **lm** capability. An explicit value of 0 (zero) indicates that the number of lines is not fixed but there is still more memory than fits on the screen.

If the terminal is one of those supported by the UNIX system virtual terminal protocol, the terminal number can be given by the **vt** capability.

Media copy strings that control an auxiliary printer connected to the terminal can be given with the following capabilities:

**ps** Prints the contents of the screen

**pf** Turns off the printer

**po** Turns on the printer

When the printer is on, all text sent to the terminal is sent to the printer. It is undefined whether the text is also displayed on the terminal screen when the printer is on. A variation, the **pO** capability, takes one parameter and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. All text, including

**pf**, is transparently passed to the printer while **pO** is in effect.

### Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The **tc** string capability can be given with the name of the similar terminal. This capability must be specified last, and the combined length of the entries must not exceed 1024. The capabilities given before **tc** override those in the terminal type invoked by **tc**.

A capability can be canceled by placing *xx***@** to the left of the **tc** invocation, where *xx* is the capability. For example, the following entry defines a Hewlett-Packard **2621-nl** that does not have the **ks** or **ke** capabilities, so it does not turn on the function key labels when in visual mode:

**hn|2621-nl:ks@:ke@:tc=2621:**

Canceling capabilities can be useful for different modes for a terminal or for different user preferences.

## NOTES

This reference page documents obsolete function that is provided only for compatibility.

Lines and columns are now stored by the kernel, as well as in the **termcap** entry. Most applications now use the kernel information primarily; the information in **termcap** is used only if the kernel does not have any information.

The total length of a single entry, excluding only escaped newlines, cannot exceed 1024.

Not all applications support all entries.

Hazeltine terminals, which do not allow **~** (tilde) characters to be displayed, should be assigned the **hz** capability.

The **nc** capability, now obsolete, was formerly needed for Datamedia terminals, which echo **\r \n** for carriage return and then ignore a following linefeed.

Terminals that ignore a linefeed immediately after an **am** wrap, such as the Concept terminals, should be assigned the **xn** capability.

If the **ce** capability is required to get rid of standout (instead of merely writing normal text on top of it), the **xs** capability should be indicated.

Teleray terminals, where tabs turn all characters moved over to blanks, should be assigned the **xt** (destructive tabs) capability. This capability is also taken to mean that it is not possible to position the cursor on top of a "magic cookie," and that to erase standout mode, it is necessary to use delete and insert line.

The Beehive Superbee terminal, which is unable to correctly transmit the ESC or **^C** characters, has the **xb** capability, indicating that the f1 key is used for ESC and f2 for **^C**. (Only certain Beehive Superbee terminals have this problem, depending on the ROM.)

Other specific terminal problems can be corrected by adding more capabilities of the form **x***x*.

The names of many of the terminals listed is this reference page are trademarks of the companies that manufacture the terminals.

## RELATED INFORMATION

Functions: **printf(3)**.

## NAME

**termios -** Describes the terminal interface for POSIX compatibility

## SYNOPSIS

**#include <termios.h>**

## DESCRIPTION

The **/usr/include/termios.h** header file contains information used by system calls that apply to terminal files. The definitions, values, and structure in this file are required for compatibility with the Institute of Electrical and Electronics Engineers (IEEE) P1003.1 Portable Operating System Interface for Computer Environments (POSIX) standard.

The general terminal interface information is contained in the **termios.h** header file. The **termios** structure in the **termios.h** header file defines the basic input, output, control, and line discipline modes. The **termios** structure contains the following fields:

**c_iflag** Describes the basic terminal input control. The possible input modes are as follows:

**BRKINT** Interrupts a signal on the break condition. If set, the break condition generates an interrupt signal and flushes both the input and output queues. This flag is initially set by default.

**ICRNL** Maps a CR character to an NL character on input. If set, a received CR character is translated into an NL character. This flag is initially not set by default.

**IGNBRK** Ignores the break condition. If set, the break condition is not put on the input queue and is therefore not read by any process. This flag is initially not set by default.

**IGNCR** Ignores the CR character. If set, a received CR character is ignored (not read). This flag is initially not set by default.

**IGNPAR** Ignores bytes with parity errors. Not supported in the current release. This flag is initially not set by default.

**INLCR** Maps newline character (NL) to carriage return (CR) on input. If set, a received NL character is translated into a CR character. This flag is initially not set by default.

**INPCK** Enables input character byte parity checking. Not supported in the current release. This flag is initially not set by default.

**ISTRIP** Strips characters. If set, valid input characters are first stripped to 7 bits; if not set, all 8 bits are processed. This flag is initially set by default.

**IXANY** Enables any character to restart output. If set, any character received restarts output. If not set, stopped output is restarted by other conventions. This flag is initially not set by default.

**IXOFF** Enables start and stop input control. If set, the system transmits a **STOP** character when the input queue is nearly full and a **START** character when enough input has been read that the queue is nearly empty again. This flag is initially set by default.

**IXON** Enables start and stop output control. If set, a received **STOP** character suspends output and a received **START** character restarts output. The **START** and **STOP** characters perform flow control functions, but they are not read. This flag is initially set by default.

**PARMRK** Marks parity errors. If set, a character with a framing or parity error that is not ignored is read as the 3-character sequence 0377, 0, *x*, where the *x* variable is the data of the character received in error. If the **ISTRIP** mode is not set, then a valid character of 0377 is read as 0377, 0377 to avoid ambiguity. If the **PARMRK** mode is not set, a framing or parity error that is not ignored is read as the null character. This flag is initially not set by default.

**c_oflag** Specifies how the system treats output. The possible output modes are as follows:

**OCRNL** Maps **CR** character to **NL** character during output. If set, mapping occurs; if not set, mapping does not occur. This flag is initially not set by default.

**ONLCR** Maps **NL** character to **CR-NL** character sequence during output. If set, mapping occurs; if not set, mapping does not occur. This flag is initially set by default.

**ONLRET** If set, the **NL** character performs the **CR** character function. If not set, the **NL** character does not perform the **CR** character function. This flag is initially not set by default.

**ONOCR** If set, no **CR** character is sent for the column 0 (zero) position. If not set, the **CR** character is sent for the column 0 (zero) position. This flag is initially not set by default.

**OPOST** If set, the remaining flag masks are interpreted as described; otherwise, characters are transmitted without change. This flag is initially set by default.

Setting **ONLRET** or **ONLCR** causes a terminal emulator to return the error [EINVAL] for the **tcsetattr( )** function call, so these flags are effectively not supported.

**c_cflag** Describes the hardware control of the terminal. In addition to the basic control modes, this field uses the following control characters:

**CLOCAL** Specifies a local line. If set, the line is assumed to have a local, direct connection with no modem control. If not set, modem control (dialup) is assumed.

**CREAD** Enables receiver. If set, the receiver is enabled. If not set, characters are not received.

**CSIZE** Specifies the number of bits per character byte. The values of **CS7** and **CS8** are recognized. The values of **CS5** and **CS6** are ignored.

**CSTOPB** Specifies the number of stop bits. If set, two stop bits are sent; if not set, only one stop bit is sent. Higher baud rates require two stop bits. (At 110 baud, for example, 2 stop bits are required.)

**CS5** Specifies a data byte of 5 bits. This value is ignored in the current release.

**CS6** Specifies a data byte of 6 bits. This value is ignored in the current release.

**CS7** Specifies a data byte of 7 bits.

**CS8** Specifies a data byte of 8 bits.

**HUPCL** Hangs up on last close. If set, the line is disconnected when the last process closes the line or when the process terminates (when the "data terminal ready" signal drops).

**PARENB** Enable parity detection. Not supported in the current release.

**PARODD** Specifies odd parity if set or even parity if not set. Not supported in the current release.

The initial hardware control value after an open is **CS8**, **CREAD**, and **HUPCL**.

**c_lflag** Controls various terminal functions. The initial value after an open is all bits clear. In addition to the basic modes, this field uses the following mask name symbols:

**ECHO** Enables echo. If set, characters are displayed on the terminal screen as they are received.

**ECHOE** Echoes erase character as **BS-SP-BS**. If **ECHOE** is set but **ECHO** is not set, the erase character is implemented as ASCII **SP-BS**.

**ECHOK** Echoes **NL** after kill.

**ECHONL** Echoes **NL**. If **ECHONL** is set, the line is cleared when a newline function is performed whether or not **ECHO** is set. This is useful for terminals that are set to local echo (also referred to as half-duplex). Unless an escape character precedes an **EOF**, the **EOF** character is not displayed. Because the ASCII **EOT** character is the default End-of-File character, this prevents terminals that respond to the **EOT** character from hanging up.

**ICANON** Enables canonical input. If set, canonical processing is enabled, which enables the erase and kill edit functions as well as the assembly of input characters into lines delimited by **NL**, **EOF**, and **EOL**.

If **ICANON** is not set, read requests are satisfied directly from the input queue. In this case, a read request is not satisfied until one of the following conditions is met: either the minimum number of characters specified by **MIN** are received, or the time-out value specified by **TIME** has expired since the last character was received. This allows bursts of input to be read, while still allowing single-character input. The **MIN** and **TIME** values are stored in the positions for the **EOF** and **EOL** characters, respectively. The time value represents tenths of a second.

**IEXTEN** Enable extended (implementation-defined) functions. Not supported in the current release.

**ISIG** Enables signals. If set, each input character is checked against the **INTR** and **QUIT** special control characters. If a character matches one of these control characters, the function associated with that character is performed. If **ISIG** is not set, checking is not done.

**NOFLSH** Disables queue flushing. If set, the normal flushing of the input and output queues associated with the quit and interrupt characters is not done.

**TOSTOP** Sends a **SIGTTOU** signal when a process in a background process group tries to write to its controlling terminal. The **SIGTTOU** signal stops the members of the process group. If job control is not supported, this symbol is ignored.

The **ICANON**, **ECHO**, **ECHOE**, **ECHOK**, **ECHONL**, and **NOFLSH** special input functions are possible only if **ISIG** is set. These functions can be disabled individually by changing the value of the control character to an unlikely or impossible value (for example, 0377 octal or 0xFF).

**c_cc** Specifies an array that defines the special control characters. The relative positions and initial values for each function are as follows:

**VEOF** Indexes the **EOF** control character (**<Ctrl-d>**), which can be used at the terminal to generate an End-of-File character. When this character is received, all characters waiting to be read are immediately passed to the program without waiting for a newline, and the **EOF** is discarded. If the **EOF** is at the beginning of a line (no characters are waiting), zero characters are passed back, which is the standard End-of-File character.

**VEOL** Indexes the **EOL** control character (**<Ctrl-@>** or ASCII null), which is an additional line delimiter that is not normally used.

**VERASE** Indexes the **ERASE** control character (**<Backspace>**), which erases the preceding character. The **ERASE** character does not erase beyond the beginning of the line (delimited by an **NL**, **EOL**, **EOF**, or **EOL2** character).

**VINTR** Indexes the **INTR** control character (**<Ctrl-Backspace>**), which sends a **SIGINT** signal to stop all processes controlled by this terminal.

**VKILL** Indexes the **KILL** control character (**<Ctrl-u>**), which deletes the entire line (delimited by an **NL**, **EOL**, **EOF**, or **EOL2** character).

**VSTART** Indexes the **START** control character (**<Ctrl-q>**), which resumes output that has been suspended by a **STOP** character. **START** characters are ignored if the output is not suspended.

**VSTOP** Indexes the **STOP** control character (**<Ctrl-s>**), which can be used to temporarily suspend output. This character is recognized during both input and output if **IXOFF** (input control) or **IXON** (output control) is set.

**VSUSP** Indexes the **SUSP** control character (**<Ctrl-z>**), which causes a **SIGTSTP** signal to be sent to all foreground processes controlled by this terminal. This character is recognized during input if **ISIG** is set. If job control is not supported, this character is ignored.

**VQUIT** Indexes the **QUIT** control character (**<Ctrl-v>** or **<Ctrl-|>**), which sends a **SIGQUIT** signal to stop all processes controlled by this terminal and writes a saveabend file into the current working directory.

The following values for the *optional_actions* parameter of the **tcsetattr( )** function are also defined in the **termios.h** header file:

**TCSADRAIN** Waits until all output written to the object file has been transmitted before setting the terminal parameters from the **termios** structure.

**TCSAFLUSH** Waits until all output written to the object file has been transmitted and all input received but not read has been discarded before setting the terminal parameters from the **termios** structure.

**TCSANOW** Immediately sets the parameters associated with the terminal from the referenced **termios** structure.

The following values for the *queue_selector* parameter of the **tcflush( )** function are also defined in the **termios.h** header file:

**TCIFLUSH** Flushes data that is received but not read.

**TCIOFLUSH** Flushes both data that is received but not read and data that is written but not transmitted.

**TCOFLUSH** Flushes data that is written but not transmitted.

The following values for the *action* parameter of the **tcflow( )** function are also defined in the **termios.h** header file:

**TCIOFF** Transmits a **STOP** character to stop data transmission by the terminal device.

**TCION** Transmits a **START** character to start or restart data transmission by the terminal device.

**TCOOFF** Suspends the output of data by the object file named in the **tcflow( )** function.

**TCOON** Restarts the output of data that was suspended by the **TCOOFF** value.

## RELATED INFORMATION

Commands: **sh(1)**.

Functions: **tcflow(3)**, **tcflush(3)**, **tcsetattr(3)**.

## STANDARDS CONFORMANCE

The HP implementation does not support the following symbolic values for the **c_oflag** field in the XPG4 Version 2 specification:

- **BSDLY**, **CRDLY**, **FFDLY**, **NLDLY**, **OFILL**, **TABDLY**, and **VTDLY**.

## NAME

**tty -** Is the general terminal interface

## SYNOPSIS

**#include <termios.h>**

## DESCRIPTION

The **tty** interface is the general interface for terminal devices. This interface supplies all the functions needed for I/O over console serial lines, workstation screens, keyboards, and other terminal devices. It consists of the special file **/dev/tty** and terminal drivers used for conversational computing.

Much of a terminal interface's performance is governed by the settings in the terminal's **termios** structure, which is defined in the **termios.h** header file. This structure provides definitions for terminal input and output processing, control and local modes, and so on.

### The Controlling Terminal

Open System Services supports the concept of a controlling terminal. Any process in the system can have a controlling terminal associated with it. Certain events, such as the delivery of keyboard-generated signals (for example, interrupt, quit, and suspend), affect all the processes in the process group associated with the controlling terminal. The controlling terminal also determines the physical device that is accessed when the indirect device **/dev/tty** is opened.

In Open System Services, in accordance with the POSIX 1003.1 specification, a process must be a session leader to allocate a controlling terminal. (This implies that the **O_NOCTTY** flag to the **open( )** function must be cleared.) The following code example illustrates the correct sequence for obtaining a controlling **tty** (no error checking is shown). This code fragment calls the **setsid( )** function to make the current process the group and session leader and to remove any controlling **tty** that the process might already have. The code then opens a terminal and attaches it to the current session as the controlling terminal. Note that the process must not already be a session or process group leader and that the console must not already be the controlling **tty** of any other session.

```
(void)setsid();         /* become session leader and */
                        /* lose controlling tty */
fd = open("/G/ZTNT/#PTY5", O_RDWR);
```

When a controlling terminal file is closed, pending input is removed and pending output is sent to the receiving device.

When a terminal file is opened, the process blocks until a carrier signal is detected. If the **open( )** function is called with the **O_NONBLOCK** flag set, however, the process does not wait. Instead, the first **read( )** or **write( )** function call waits for a carrier to be established. If the **CLOCAL** mode is set in the **termios** structure, the driver assumes that modem control is not in effect and so the **open( )**, **read( )**, and **write( )** calls proceed without waiting for a carrier signal to be established.

### Process Groups

Each OSS process belongs to a process group with a specific process group ID. Each OSS process belongs to the process group of its creating process. This enables related processes to be signaled. Process group IDs are unique identifiers that cannot be used for other system process groups until the original process group is disbanded. Each process group also has a process group leader. A process group leader has the same process ID as its process group.

Each process group belongs to a session. Each process in the process group also belongs to the process group's session. A process that is not the process group leader can create its own session and process group with a call to the **setsid( )** function. That calling process then becomes the session leader of the new session and of the new process group. The new session has no controlling terminal until the session leader assigns one to it. The calling process's ID is assigned to the new process group. With the **setpgid( )** function, other processes can be added to a process group.

A controlling terminal can have a process group associated with it that is known as the foreground process group. The terminal's foreground process group is the one that receives signals generated by the **VINTR**, **VQUIT**, and **VSUSP** special control characters. Certain operations on the terminal are also restricted to processes in the terminal's foreground process group (see **Terminal Access Control**, later in this reference page). A terminal's foreground process group can be changed by calling the **tcsetpgrp()** function. A terminal's current foreground process group can be obtained by calling the **tcgetpgrp()** function.

**Input Processing Modes**

The terminal drivers have two major modes, characterized by the kind of processing that takes place on the input characters:

**Canonical** If a terminal is in canonical mode, input is collected and processed one line at a time. Lines are terminated by a newline (*L0), End-of-File (**VEOF**), or End-of-Line (**EOL**) character. A read request is not returned until either the line is terminated or a signal is received. The maximum number of bytes of unread input allowed on an input terminal is 255 bytes.

Erase and kill processing is performed on input that was not terminated by one of the line-termination characters. Erase processing removes the last character in the line; kill processing removes the whole line.

**Noncanonical** Noncanonical mode eliminates erase and kill processing, making input characters available to the user program as they are typed. Input is not processed into lines. The received bytes are processed according to the **VMIN** and **VTIME** elements of the **c_cc** array in the **termios** structure.

**VMIN** The minimum number of bytes the terminal can receive in noncanonical mode before a read is considered successful.

**VTIME** Measured in 0.1-second units, times out sporadic input.

These cases are summarized as follows:

**VMIN**>0, **VTIME**>0

In this case, **VTIME** is an interbyte timer that is activated after the first byte of the input line is received and reset after each byte is received. The read operation is a success if **VMIN** bytes are read before **VTIME** runs out. If **VTIME** runs out before **VMIN** bytes are received, the characters that were received are returned.

**VMIN**>0, **VTIME**=0

In this case, only **VMIN** is used. A queued **read( )** function call waits until either **VMIN** bytes are received or a signal is received.

**VMIN**=0, **VTIME**>0

In this case, **VTIME** is used as a read timer that starts when a **read( )** function call is made. The **read( )** call is finished either when one byte is read or when **VTIME** runs out.

**VMIN**=0, **VTIME**=0
In this case, either the number of requested bytes or the number of currently available bytes is returned, depending on which is less. The **read( )** function call returns 0 (zero) if no data was read.

Canonical mode is entered by setting the **ICANON** flag of the **c_lflag** field in the terminal's **termios** structure. Other input processing is performed according to the other flags set in the **c_iflag** and **c_lflag** fields.

**Input Editing**

A terminal ordinarily operates in full-duplex mode. Characters can be typed at any time, even while output is occurring. Characters are lost only when the system's character input buffers become completely overrun, which is rare, or when the user has accumulated the maximum allowed number of input characters (**MAX_INPUT/2**) that have not yet been read by some program. Currently this limit is 255 characters.

Input characters are normally accepted in either even or odd parity with the parity bit being stripped off before the character is given to the program. The **ISTRIP** mask of the **c_iflag** field controls whether the parity bit is stripped (**ISTRIP** set) or not stripped (**ISTRIP** not set). By setting the **PARENB** flag in the **c_cflag** field and either setting (or not setting) the **PARODD** flag, it is possible to have input characters with even or odd parity discarded or marked (see **Input Modes**, later in this reference page).

Input characters are normally echoed by putting them in an output queue as they arrive. This can be disabled by clearing the **ECHO** bit in the **c_lflag** word using the **tcsetattr( )** function.

In canonical mode, terminal input is processed in units of lines. A program attempting to read is normally suspended until an entire line is received (however, see the description of the **SIGTTIN** signal under **Terminal Access Control**, later in this reference page). No matter how many characters are requested in the read call, at most one line is returned. It is not, however, necessary to read a whole line at once; any number of characters can be requested in a read, even one, without losing information. In **read( )** requests, the **O_NONBLOCK** flag affects the read operation.

If the **O_NONBLOCK** flag is not set, a **read( )** request is blocked until either data or a signal is received. If the **O_NONBLOCK** flag is set, the **read( )** request is not blocked and one of the following situations occurs:

- Some data might have been typed, but there might or might not be enough data to satisfy the entire **read( )** request. In either case, the **read( )** function returns the data available, returning the number of bytes of data it read.
- If there is no data for the read operation, the **read( )** function returns the value **-**1 with an error value of [EAGAIN].

During input, line editing is normally done with the erase special control character (**VERASE**) logically erasing the last character typed and the kill special control character (**VKILL**) logically erasing the entire current input line. These characters never erase beyond the beginning of the current input line or an End-of-File (**VEOF**) character. These characters, along with the other special control characters, can be entered literally by preceding them with the literal-next character (**VLNEXT**, for which the default value is **<^V>**).

The drivers normally treat either a newline character (**\n**), End-of-File character (**VEOF**), or End-of-Line character (**VEOL**) as terminating an input line, echoing a return and a linefeed. If the **ICRNL** mode is set in the **c_iflag** word of the **termios** structure, then carriage returns are translated to newline characters on input and are normally echoed as carriage return-linefeed sequences. If **ICRNL** is not set, this processing for carriage return is disabled; it is simply echoed as a return and does not terminate canonical mode input.

### Input Modes

The **termios** structure has an input mode field **c_iflag**, which controls basic terminal input characteristics. These characteristics are masks that can be bitwise inclusive ORed. The masks include:

| | |
|---|---|
| **BRKINT** | An interrupt is signaled on a break condition. |
| **ICRNL** | All carriage returns are mapped to newline characters when input. |
| **IGNBRK** | Break conditions are ignored. |
| **IGNCR** | Carriage returns are ignored. |
| **INLCR** | Newline characters are mapped to carriage returns when they are input. |
| **ISTRIP** | The 8th bit (parity bit) is stripped on input characters. |
| **IXOFF** | **STOP** and **START** characters are sent to enable input flow control. |
| **IXON** | **STOP** and **START** characters are recognized for output flow control. |
| **PARMRK** | Parity errors are marked with a 3-character sequence. |

The input mode mask bits can be combined for the following results.

- If **IGNBRK** is set, input break conditions are ignored. If **IGNBRK** is not set but **BRKINT** is set, the break condition has the same effect as if the **VINTR** control character had been typed; that is, a **SIGINT** signal is generated. If neither **IGNBRK** nor **BRKINT** are set, then the break condition is input as a single character 0x00 (zero). If the **PARMRK** flag is also set, then the input is read as three characters, 0xff, 0x00, and 0x00.
- If **PARMRK** is set, a byte with a parity or framing error, except for breaks, is passed as the three characters 0xff, 0x00, and *X*, where *X* is the character data received in error. If **ISTRIP** is not set, the valid character 0xff is passed as 0xff, 0xff. If **PARMRK** is not set, framing or parity errors, including breaks, are passed as the single character 0x00.
- Setting **ISTRIP** causes the 8th bit of the 8 valid input bits to be stripped before processing. If **ISTRIP** is not set, all 8 bits are processed.
- Setting **INLCR** causes a newline character to be read as a carriage return character. If **IGNCR** is also set, the carriage return is ignored. If **IGNCR** is not set, **INLCR** works as described earlier in **Input Modes**.
- The **STOP** character (normally **<Ctrl-S>**) suspends output, and the **START** character (normally **<Ctrl-Q>**) restarts output. Setting **IXON** enables stop/start output control, in which the **START** and **STOP** characters are not read but rather perform flow control functions. Extra **STOP** characters typed when output is already stopped have no effect, unless the **START** and **STOP** characters are made the same, in which case output resumes. If **IXON** is not set, the **START** and **STOP** characters are read.
- If **IXOFF** is set, stop/start input control is enabled. When **IXOFF** is set, the terminal device is sent **STOP** characters to halt the transmission of data when the input queue is in danger of overflowing (exceeds the size **MAX_INPUT**/2). When enough characters are read to reduce the amount of data queued to an acceptable level, a **START** character is sent to the device to allow it to continue transmitting data. This mode is useful when the terminal is actually another machine that obeys these conventions.

**Input Echoing and Redisplay**

The terminal driver has several modes for handling the echoing of terminal input, controlled by bits in the **c_lflag** field of the **termios** structure.

**Erasing Characters From a CRT**

When a CRT terminal is in use, the **ECHOE** bit of the **c_lflag** field of the **termios** structure can be set to cause input to be erased from the screen with a backspace-space-backspace sequence when character-deleting or word-deleting sequences are used.

**Output Processing**

When one or more characters are written, they are actually transmitted to the terminal as soon as previously written characters have finished typing. Input characters are normally echoed by putting them in the output queue as they arrive. When a process produces characters more rapidly than the terminal can accept them, it is suspended when its output queue exceeds some limit. When the queue has come down to some threshold, the program resumes.

**Line Control and Breaks**

The **tcsendbreak( )** function can cause a break condition for a specified amount of time. Break conditions in the input are handled according to the value in the **c_iflag** field of the **termios** structure. (Refer to **Input Modes**, earlier, for a complete list of the **c_iflag** field settings.)

When a TELNET disconnect is detected, all OSS open file descriptors are cleared if the terminal window is a dynamic window or a static window with **CLOCAL** not set in the **c_cflag** field of the **termios** structure. All outstanding write requests fail with the error [EIO]. All outstanding read requests return zero bytes read. If **CLOCAL** is set on a static window, outstanding read and write requests are queued until a new TELNET connection is established. If **CLOCAL** is not set and a static window is a controlling terminal, a **SIGHUP** signal is sent to the window's controlling process.

**Interrupt Characters**

When **ISIG** is set in the **c_lflag** word of **termios**, there are several characters that generate signals in both canonical and noncanonical mode; all are sent to the processes in the foreground process group of the terminal. If **NOFLSH** is not set in **c_lflag**, these characters also flush pending input and output when typed at a terminal. The characters shown here are the default characters; the symbolic names of the indexes of these characters in the **c_cc** array of **termios** are also shown. The characters are as follows:

**^C** — **VINTR** (in **c_cc**) generates a **SIGINT** signal. This is the normal way to stop a process or to regain control in an interactive program.

**^** — **VQUIT** (in **c_cc**) generates a **SIGQUIT** signal. This causes a program to terminate and produce a saveabend file, if possible, in the current directory.

**^Z** — **VSUSP** (in **c_cc**) generates a **SIGTSTP** signal, which is used to suspend the current process group.

**Terminal Access Control**

If a process attempts to read from its controlling terminal when the process is not in the foreground process group of the terminal, that background process group is sent a **SIGTTIN** signal, the read returns a -1, and **errno** is set to [EINTR]. This signal normally causes the members of that process group to stop. If, however, the process is ignoring **SIGTTIN** or has **SIGTTIN** blocked, or if the reading process's process group is orphaned, the read returns the value -1 with **errno** set to [EIO] and does not send a signal.

If a process attempts to write to its controlling terminal when the process is not in the foreground process group of the terminal, and if **TOSTOP** is set in the **c_lflag** word of the **termios** structure, the background process group is sent a **SIGTTOU** signal and the process is prohibited from writing. If **TOSTOP** is not set, or if **TOSTOP** is set and the process is blocking or ignoring the

**SIGTTOU** signal, the writes to the terminal are allowed and the **SIGTTOU** signal is not sent. If **TOSTOP** is set, if the writing process's process group is orphaned, and if **SIGTTOU** is not blocked by the writing process, the write operation returns the value -1 with **errno** set to [EIO] and does not a send a signal.

### The tty Parameters

In contrast to earlier versions of the **tty** driver, the OSS terminal parameters and structures are contained in a single structure, which is the **termios** structure defined in the **termios.h** file.

### Basic System Calls

A large number of system calls apply to terminals. The applicable calls follow:

**tcgetattr( )** Gets the **termios** structure and all of its associated parameters. The interface delays until output is quiescent, then throws away any unread characters.

**tcsetattr(TCSANOW)**
Immediately sets the parameters according to the **termios** structure.

**tcsetattr(TCSADRAIN)**
Waits until all output is transmitted and input is read before setting the parameters according to the **termios** structure.

**tcsetattr(TCSAFLUSH)**
Waits until all output is transmitted before setting the parameters according to the **termios** structure. Discards all unread input before setting the parameters.

**tcflush( )** Flushes unread input data, nontransmitted output data, or both.

The following system calls perform miscellaneous functions on the controlling terminal. In cases where arguments are required, they are described as a parameter named *arg*. Otherwise, *arg* should be specified as the value 0 (zero).

**tcflow(TCIOFF)**
Output is stopped as if the **STOP** character were typed.

**tcflow(TCION)**
Output is restarted as if the **START** character were typed.

**tcflow(TCOOFF)**
Output is suspended.

**tcflow(TCOON)**
Suspended output is restarted.

**tcgetpgrp( )** The *arg* parameter is a pointer to an **int** parameter into which is placed the process group ID of the process group for which this terminal is the control terminal.

**tcsetpgrp( )** The *arg* parameter is a pointer to an **int** parameter containing the value to which the process group ID for this terminal will be set.

## FILES

**/dev/tty** Special file for a tty device.

## RELATED INFORMATION

Functions: **tcdrain(3)**, **tcflush(3)**, **tcgetattr(3)**, **tcgetpgrp(3)**, **tcsendbreak(3)**, **tcsetattr(3)**, **tcsetpgrp(3)**.

Commands: **sh(1)**.

Files: **termios(4)**.

# Section 12.  Miscellaneous

This section contains reference pages for some miscellaneous Open System Services (OSS) topics.  These reference pages reside in the **cat5** directory and are sorted alphabetically by U.S. English conventions in this section.

## NAME

**acl -** Introduction to OSS access control lists (ACLs)

## DESCRIPTION

Access control lists (ACLs) are a key enforcement mechanism of discretionary access control (see "Definitions" later in this reference page). ACLs specify access to files by users and groups more selectively than traditional UNIX mechanisms.

OSS already enables nonprivileged users or processes, such as file owners, to allow or deny other users access to files and other objects as determined by their user identity, group identity, or both. This level of control is accomplished by setting or manipulating a file's permission bits to grant or restrict access by owner, group, and others (see the **chmod(2)** reference page).

ACLs offer a greater degree of selectivity than permission bits. ACLs allow a process whose effective user ID matches the file owner, super ID, or a member of the Safeguard SECURITY-OSS-ADMINISTRATOR security group to permit or deny access to a file to a list of specific users and groups.

ACLs are supported as a superset of the UNIX operating system discretionary access control (DAC) mechanism for files, but not for other objects such as interprocess communication (IPC) objects.

All OSS system calls that include pathnames are subject to the ACLs on any directory or file in the path.

OSS ACLs:

- Are supported in Version 3 catalog OSS filesets on J-series RVUs, on H06.08 and later H-series RVUs, and G06.29 and later G-series RVUs.
- Are supported for directories, regular files, first-in, first-out (FIFO) special files, and bound AF_UNIX sockets.
- Support up to 150 ACL entries.
- Support separate permissions for up to 146 additional users and groups.
- Support default ACL inheritance (see "ACL Inheritance" later in this reference page).
- Are based on the POSIX 1003.1e draft standard and the HP-UX implementation of ACLs.
- Are not supported by the OSS Network File System (NFS) for J06.08 and earlier J-series RVUs, H06.19 and earlier H-series RVUs, or G-series RVUs. All attempts by NFS clients to access OSS objects protected by ACLs that contain optional ACL entries are denied.
- Are supported by the OSS NFS for J06.09 and later J-series RVUs and H06.20 and later H-series RVUs as follows:
  - Access by OSS NFS clients to OSS objects protected by optional ACL entries can be allowed, depending upon the NFSPERMMAP attribute value for the OSS fileset that contains the object.
  - The NFSPERMMAP attribute value specifies the algorithm used to map the OSS ACL permissions for the object to the standard permissions bits (rwxrwxrwx) expected for the object by NFS V2 clients.

— The default value for the NFSPERMMAP attribute, DISABLED, specifies that all attempts by NFS clients to access OSS objects protected by ACLs that contain optional ACL entries are denied. This behavior matches the behavior for J06.08 and earlier J-series RVUs, H06.19 and earlier H-series RVUs, and G-series RVUs.

For more information about NFS and ACLs, see "OSS Network File System (NFS) and ACLs" later in this reference page.

**Definitions**

Control of access to data is a key concern of computer security. These definitions, based on the *Department of Defense Trusted Computer System Evaluation Criteria*, explain the concepts of access control and its relevance to OSS security features:

access
: A specific type of interaction between a subject and an object that results in the flow of information from one to the other. Subjects include persons, processes, or devices that cause information to flow among objects or change the system state. Objects include files (ordinary files, directories, special files, FIFOs, and so on) and IPC features (shared memory, message queues, semaphores, and sockets).

access control list (ACL)
: An access control list is a set of *user.group, mode* entries associated with a file that specifies permissions for all possible combinations of user IDs and group IDs.

access control list (ACL) entry
: An entry in an ACL that specifies access rights for a file owner, owning group, group class, additional user, additional group, or all others.

change permission
: The right to alter DAC information (permission bits or ACL entries). Change permission is granted to object (file) owners and to privileged users.

discretionary access control (DAC)
: A means of restricting access to objects based on the identity of subjects, groups to which they belong, or both. The controls are discretionary because a subject with a certain access permission is able to pass that permission (perhaps indirectly) to any other subject.

mode
: Three bits in each ACL entry that represent read, write, and execute or search permissions.

privilege
: The ability to ignore access restrictions and change restrictions imposed by security policy and implemented in an access control mechanism. In OSS, the super ID is the only user ID that can ignore access restrictions. However, the super ID and any member of the Safeguard SECURITY-OSS-ADMINISTRATOR security group can change the ownership and access permissions (standard UNIX permissions or ACL entries) of a file.

**Access Control List Entries**

An ACL consists of a set of one-line entries that specify permissions for a file. Each entry specifies for one user-ID or group-ID a set of access permissions, including read, write, and execute/search.

To understand the relationship between access control lists and traditional file permissions, consider the following file and its permissions:

**-rwxr-xr-- james admin datafile**

For this file:

- The owner is the user **james**.
- The group is **admin**.
- The name of the file is **datafile**.
- The file owner permissions are **rwx**.
- The file group permissions are **r-x**.
- The file other permissions are **r--**.

In an ACL, user and group IDs are represented by names or numbers, as found in the user authentication database and group database for the system.

## ACL Notation

Supported commands that manage ACLs recognize these symbolic representations:

**[d[efault]:]u[ser]:[**_uid_**]:**_perm_
**[d[efault]:]g[roup]:[**_gid_**]:**_perm_
**[d[efault]:]c[lass]:**_perm_
**[d[efault]:]o[ther]:**_perm_

An ACL entry prefixed with **d:** or **default:** can only occur in ACLs for directories. The prefix indicates that the remainder of the entry is not to be used in determining the access rights to the directory but is instead to be applied to any files or subdirectories created in the directory (see "ACL Inheritance" later in this reference page).

The *uid* and *gid* fields contain either numeric user or group IDs, or their corresponding character strings from the authentication database and group database for the system.

The *perm* field indicates access permission either in symbolic form, as a combination of **r**, **w**, **x**, and - (dash), or in numeric form, as an octal value of 0 through 7 representing the sum of 4 for read permission, 2 for write permission, and 1 for execute permission.

## Types of ACL Entries

An ACL can contain several types of entries:

Base ACL Entries

The base ACL entries grant permissions equivalent to standard UNIX permissions. When an ACL consists of the four base ACL entries only, it is called a minimal ACL, and the permissions for the **class** and **other** ACL entries are equal. The **chmod( )** and **acl( )** functions can change base ACL entries. Base ACL entries are:

| Notation | Entry Type | Description |
|---|---|---|
| **user::**_perm_ | USER_OBJ | Permissions for the owner of the object |
| **group::**_perm_ | GROUP_OBJ | Permissions for the owning group of the object |
| **class:**_perm_ | CLASS_OBJ | The maximum permissions granted to the file group class |
| **other:**_perm_ | OTHER_OBJ | Permissions for other users |

Class Entry

The **class** entry, which is one of the base ACL entries, acts as an upper bound for file permissions. In an ACL that contains optional group entries or optional user entries, the **class** entry specifies the maximum permissions that can be granted to:

- Members of the owning group
- Any additional **user** entries (optional users)
- Any additional **group** entries (members of any optional groups)

The **class** entry is useful because it allows you to restrict the permissions for all of the other ACL entries by changing only one ACL entry. If optional **user** or optional **group** ACL entries are present, the **chmod** command changes the permissions of the **class** ACL entry instead of the permissions of the owning group. This behavior allows programs that use the **chmod** command to support files or directories that have permissions for additional users and groups.

Optional ACL Entries

Optional ACL entries are ACL entries other than the base ACL entries. Optional ACL entries grant permissions beyond the standard UNIX permissions and can be used to further allow or deny access to the file. A file or directory is considered to "have an ACL" only if optional ACLs are present. In OSS, you can specify up to 146 optional ACL entries in an ACL. You use the **setacl** command or the **acl( )** system call to set ACL entries. Nondefault optional ACL entries include:

| Notation | Entry Type | Description |
|---|---|---|
| **user:***uid***:***perm* | USER | Permissions for the user specified by *uid* |
| **group:***gid***:***perm* | GROUP | Permissions for the group specified by *gid* |

Examples of nondefault optional ACL entries:

**u:mary:rwx**

Grant read, write, and execute access to the user **mary**.

**user:george:---**

Deny access to the user **george**.

**g:writers:rw-**

Grant read and write access to all members of the group **writers**.

Actual ACL Entries

The base ACL entries, optional user ACL entries, and optional group ACL entries are considered "actual" ACL entries because they actually control access to the associated file or directory. These ACL entries are also called nondefault ACL entries. Contrast with "Default ACL Entries."

Default ACL Entries

Default ACL entries are allowed for directories only. Default ACL entries do not determine who can access the directory. Instead, default ACL entries affect the access permissions for files or directories created in the directory (see "ACL Inheritance" later in this reference page). All default ACL entries are optional ACL entries. Default ACL entries include:

| Notation | Entry Type | Description |
|---|---|---|
| **default:user::***perm* | DEF_USER_OBJ | Default permissions for the object owner |
| **default:user:***uid***:***perm* | DEF_USER | Default permissions for additional users specified by *uid* |
| **default:group::***perm* | DEF_GROUP_OBJ | Default permissions for members of the owning group of the object |
| **default:group:***gid***:***perm* | DEF_GROUP | Default permissions for members of the additional group specified by *gid* |
| **default:class:***perm* | DEF_CLASS_OBJ | Default maximum permissions granted to the file group class |
| **default:other:***perm* | DEF_OTHER_OBJ | Default permissions granted to other users |

These entries are sometimes referred to as base default ACL entries because the permissions for these entries in the parent directory, modified by the file-creation mode, the umask, or both, become the permissions for the base ACL entries for a new file when the new file inherits ACL entries from the parent directory:

- **default:user::***perm* (DEF_USER_OBJ)
- **default:group::***perm* (DEF_GROUP_OBJ)
- **default:class:***perm* (DEF_CLASS_OBJ)
- **default:other:***perm* (DEF_OTHER_OBJ)

**ACL Uniqueness**

Entries are unique in each ACL. An ACL can contain only one of each type of base entry, and one entry for any given user or group ID. Likewise, an ACL can contain only one of each type of default base entry, and one default entry for any given user or group ID.

**ACL Inheritance**

The permissions, including access control list entries, if any, for a newly created file are determined by:

- Whether the fileset of the created file supports OSS ACLs
- Whether the system on which the process is running supports OSS ACLs
- Whether the parent directory of the created file contains default ACL entries
- The file-creation mode (*mode*)
- The process umask (*umask*)

If the fileset does not support OSS ACLs, the permissions of the created file are the *mode* bitwise-ANDed with the complement of the *umask*.

If the fileset supports OSS ACLs, but the system on which the process is running does not support OSS ACLs, and the parent directory for the created file does not have default ACL entries, the permissions of the created file are the *mode* bitwise-ANDed with the complement of the *umask*.

If the fileset supports OSS ACLs, but the system on which the process is running does not support OSS ACLs, and the parent directory for the created file has default ACL entries:

- The permissions for the base ACL entries of the created file or directory are determined by a combination of the file-creation mode and the default base ACL entries of the parent directory as follows:

USER_OBJ permissions
: The DEF_USER_OBJ permissions bitwise-ANDed with the *mode* **user** permissions bitwise-ANDed with the complement of the *umask* **user** permissions.

GROUP_OBJ permissions
: The DEF_GROUP_OBJ permissions

CLASS_OBJ permissions
: The DEF_CLASS_OBJ permissions bitwise-ANDed with the *mode* **group** permissions bitwise-ANDed with the complement of the *umask* **group** permissions

OTHER_OBJ permissions
: The DEF_OTHER_OBJ permissions bitwise-ANDed with the *mode* **other** permissions bitwise-ANDed with the complement of the *umask* **other** permissions

- The default optional ACL entries for the parent directory of the created file are added to the ACL of the created file as actual (nondefault) optional ACL entries.
- If the created file is a directory, all of the default ACL entries of the parent directory are added to the ACL of the new directory. This behavior allows ACL entries to be inherited by files and directories created under this new directory.

If both the fileset for the created file and the system in which the process is running support OSS ACLs, and the parent directory for the created file does not have default ACL entries, the permissions of the created file are the *mode* bitwise-ANDed with the complement of the *umask*.

If both the fileset for the created file and the system in which the process is running support OSS ACLs, and the parent directory of the created file contains default ACL entries:

- The permissions for the base ACL entries of the created file or directory are determined by a combination of the file-creation mode and the default base ACL entries of the parent directory as follows:

  USER_OBJ permissions
  : The DEF_USER_OBJ permissions bitwise-ANDed with the *mode* **user** permissions

  GROUP_OBJ permissions
  : The DEF_GROUP_OBJ permissions

  CLASS_OBJ permissions
  : The DEF_CLASS_OBJ permissions bitwise-ANDed with the *mode* **group** permissions

  OTHER_OBJ permissions
  : The DEF_OTHER_OBJ permissions bitwise-ANDed with the *mode* **other** permissions

- The default optional ACL entries for the parent directory of the created file are added to the ACL of the created file as actual (nondefault) optional ACL entries.
- If the created file is a directory, all of the default ACL entries of the parent directory are copied to the ACL of the new directory. This behavior allows default ACL entries to be inherited by files and directories created under this new directory.

For security reasons, if an ACL contains default ACL entries, all of the default base ACL entries should be provided. During ACL inheritance, if any default base ACL entries are missing, the permissions for the missing default base ACL entries are derived as follows:

DEF_USER_OBJ permissions
: The complement of the *umask* **user** permissions

DEF_GROUP_OBJ permissions
: The complement of the *umask* **group** permissions

DEF_CLASS_OBJ permissions
: The complement of the *umask* **group** permissions

DEF_OTHER_OBJ permissions
: The complement of the *umask* **other** permissions

### Examples of ACL Inheritance

Directory **/a** has the following ACL, as reported by the **getacl** command:

```
# file: /a
# owner: alpha
# group: uno
user::rwx
group::rwx
class:rwx
other:rwx
default:user:beta:r--
default:user:gamma:r--
default:group:dos:---
default:group:tres:---
```

In this example, the ACL for a new file created in the directory **/a** includes the default ACL entries for directory **/a** as actual (nondefault) ACL entries:

# file: /a/newfile
# owner: *creator_uid*
# group: *creator_gid*
user::rw-
user:beta:r--
user:gamma:r--
group::r--
group:dos:---
group:tres:---
class:r--
other:r--

In this example, a new directory, **dir** is created in the **/a** directory. The default ACL entries of the parent directory, **/a**, are added to the ACL of the new subdirectory twice, first as actual (nondefault) ACL entries and second as the default ACL entries. This behavior ensures that default ACLs propagate downward as trees of directories are created. This example shows the ACL of the new directory, **dir**:

# file: /a/dir
# owner: *creator_uid*
# group: *creator_gid*
user::rwx
user:beta:r--

user:gamma:r--
group::r-x
group:dos:---
group:tres:---
class:r-x
other:r-x
default:user:beta:r--
default:user:gamma:r--
default:group:dos:---
default:group:tres:---

### Access Check Algorithm

To determine the permissions granted to an accessing process, the operating system checks for matching IDs in the following order:

1. If the EUID of the process is the same as the owner of the file, grant the permissions specified in the **user::** entry of the ACL. Otherwise, continue to the next check.

2. If the EUID matches the UID specified in one of the additional **user:uid:** ACL entries, grant the permissions specified in that entry bitwise-ANDed with the permissions specified in the class entry. Otherwise, continue to the next check.

3. If the EGID or a supplementary GID of the process matches the owning GID of the file or one of the GIDs specified in any additional **group:gid:** ACL entries, grant the permissions specified in the class entry bitwise-ANDed with the result of bitwise-ORing together all of the permissions in all matching group entries. Otherwise, continue to the next check.

4. Grant the permissions specified in the **other:** ACL entry.

Because the checks are performed in this order and the ID match checking stops when a match is found, you can use optional user or group ACL entries with restrictive permissions to deny access to certain users or groups.

If the EGID, the supplementary GIDs of the process, or both match the GIDs of multiple group ACL entries for a file, the process is granted the permissions of all of the matching group entries, restricted by the permissions in the class entry. For example, assume that the effective user ID for a process represents the user **beta**, and the group IDs for that process represent group membership only in the **dos** and **tres** groups. In this example, that process is allowed to open the file **/a/file** with read/write access, because the **group:dos:** entry granted read access, the **group:tres:** entry granted write access, and the **class:** entry allowed read and write access.

\# file: /a/file
\# owner: *creator_uid*
\# group: *creator_gid*
user::rw-
group::rw-
group:dos:r--
group:tres:-w-
class:rw-
other:r--

### ACL Operations Supported

- The **acl( )** system call sets, retrieves, or counts ACLs.
- The **setacl** command sets or modifies ACLs.
- The **getacl** command retrieves ACLs.
- The **-acl** option of the **find** command locates files with certain ACL properties.
- The **cp**, **cpio -p**, **mv**, and **pax -rw** commands copy ACLs with the source files to the destination files.
- The Backup and Restore 2 utility backs up ACLs with the files on tape and restores ACLs with the files back to disk.

### ACL Interaction with stat( ), chmod( ), and chown( )

**stat( )**
The *st_mode* field summarizes the access rights to the file. It differs from file permission bits only if the file has one or more optional ACL entries. If one or more optional ACL entries are present in the ACL of the file, the permissions specified in the class entry of the ACL are returned as the permissions for group in the *st_mode* field. Because of this behavior, programs that use the **stat( )** or **chmod( )** functions and are ignorant of ACLs are more likely to produce expected results. The *st_acl* field indicates the presence of optional ACL entries in the ACL for the file. The *st_basemode* field provides the owning user permissions, owning group permissions, and other permissions for the file.

**chmod( )**
Using the **chmod( )** function to set the group permission bits affects the **class:** entry for the file, which in turn affects the permissions granted by additional **user:uid:** and **group:gid:** entries. In particular, using **chmod( )** to set file permission bits to all zeros removes all access to the file, regardless of permissions granted by any additional **user:uid:** or **group:gid:** entries. If the **chmod( )** function is used on an object that does not have optional ACL entries, both the **class** ACL entry and the owning group ACL entry permission bits are changed to the new group permissions value.

**chown( )**
If you use the **chown( )** function to change the owner or owning group of a file to a user ID or group ID that has an existing **user:uid:** or **group:gid:** entry in the ACL for the file, those existing entries are not removed from the ACL. However, those existing entries no longer have any effect, because the **user::** or **group::** entries take precedence.

### OSS Network File System (NFS) and ACLs

For J06.09 and later J-series RVUs and H06.20 and later H-series RVUs, access by the OSS Network File System (NFS) to OSS objects protected by ACLs that contain optional ACL entries can be allowed, depending upon the NFSPERMMAP attribute value for the fileset that contains the object.

The NFSPERMMAP attribute is an attribute of the OSS fileset and is set using Subsystem Control Facility (SCF) commands. For information about OSS SCF commands, see the *Open System Services Management and Operations Guide*.

NFS Version 2 (NFS V2) clients make their own access decisions based on their interpretation of the permissions bits of the object. Because NFS Version 2 does not support ACLs, the ACL entries must be mapped to the nine basic permissions bits (rwxrwxrwx) used for objects in NFS. An object that is protected by an ACL cannot reflect the correct access for all users in these nine permission bits. It may be that access that would be granted by the mapped permission bits is actually denied explicitly by the ACL. It may also be that access that seems to be denied by the mapped permission bits is, in fact, granted explicitly by the ACL.

The value of the NFSPERMMAP attriute specifies how the permissions for an OSS object protected by optional access control list (ACL) entries are mapped to the standard permissions bits (rwxrwxrwx) used by NFS V2 clients on open, read, write, and directory search operations. Write permissions are always enforced on the NonStop server using the actual standard OSS permissions or OSS ACL permissions (if present) on the object. The values for the NFSPERMMAP attribute are:

**RESTRICTIVE**
The other and owning group fields of the permissions bits returned to NFS V2 clients are modified such that only access that would be granted to **everyone** in the ACL, excluding the owner, is granted in the permissions bits. That is:

- The ACL entries for the class mask, the owning group, and all optional users are examined. The group permissions returned to NFS V2 clients for this object are the **most restrictive** of the permissions bits of these ACL entries.
- The ACL entries for the class mask, the owning group, **other**, all optional groups, and all optional users are examined. The other permissions returned to NFS V2 clients for this object are the **most restrictive** of the permissions bits of these ACL entries.

Setting NFSPERMAP to this value can cause some users on NFS V2 clients to be denied access to objects to which they should legitimately be granted acceses according to the OSS ACL on the NonStop server.

**PERMISSIVE** The other and owning group fields of the permissions bits returned to NFS V2 clients are modified such that access that would be granted to **anyone** in the ACL, excluding the owner, is granted in the permissions bits. That is:

- The ACL entries for the class mask, the owning group, and all optional users are examined. The group permissions returned to NFS V2 clients for this object are the **most permissive** of the permissions bits, as allowed by the class mask, of these ACL entries.
- The ACL entries for the class mask, the owning group, **other**, all optional groups, and all optional users are examined. The other permissions returned to NFS V2 clients for this object are the **most permissive** of the permissions bits for the **other** ACL entry and, as allowed by the class mask, the ACL entries of the owning group, optional groups, and optional users.

Setting NFSPERMMAP to this value guarantees that users who have read permission in the OSS ACL for an object on the NonStop system will be able to read the object on NFS V2 clients. However, it also allows users on the NFS V2 client who do not have read permission in the OSS ACL for an object on the NonStop Server to be able to read data from the object when the data is cached on the NFS V2 client.

**UNMODIFIED**
The other and user fields of the permissions bits returned to NFS V2 clients are unmodified. The group field of the permissions bits returned to NFS V2 clients are the permissions of the **class** entry of the ACL. This set of permissions bits matches the permissions that are displayed on the NonStop server by a command such as the **ls** command.

**DISABLED** Disables the mapping of OSS ACLs to NFS file permissions. When NFSPERMMAP is disabled, NFS requests to objects protected by OSS ACLs that contain optional ACL entries are denied. This behavior matches the behavior for systems running J06.08 and earlier J-series RVUs, H06.19 and earlier H-series RVUs, and G-series RVUs. This is the default value.

To demonstrate the effect of the value of NFSPERMMAP attribute on the permissions returned to NFS V2 clients, consider this file:

```
> setacl -m g:GRP1:--x myfile1
> getacl myfile1
# file: myfile1
# owner: SUPER.SUPER
# group: SUPER
user::rw-
user:TEST.USER01:--x
user:SUPER.USER01:-w-
group::r--
group:TEST1:-w-
group:GRP1:--x
class:rwx
other:rw-
```

The ACL for the file **myfile1** has two optional user entries and two optional group entries. The permissions returned to the OSS NFS V2 clients are as follows:

- If the NFSPERMMAP attribute is set to **RESTRICTIVE**, the permissions returned are: **rw-------**.
- If the NFSPERMMAP attribute is set to **PERMISSIVE**, the permissions returned are: **rw-rwxrwx**.
- If the NFSPERMMAP attribute is set to **UMODIFIED**, the permissions returned are: **rw-rwxrw-**.
- If the NFSPERMMAP attribute is set to **DISABLED**, all OSS NFS V2 clients are denied access to this file.

In the example, a user in the group **TEST1** is allowed write access to **myfile1** if that user accesses the file using the OSS filesystem. But, if NFSPERMMAP is **RESTRICTIVE**, and that user tries to access **myfile1** using the NFS V2 client, that user is denied access to the file.

In contrast, if NFSPERMMAP is **PERMISSIVE** the permissions returned for **myfile1** indicate that user TEST.USER01 has permission to write to the file. However, because the ACL for the file does not grant write permission to TEST.USER01, attempts to open the file might succeed but attempts to write to the file fail with the [EACCESS] error because all write permissions are enforced on the NonStop server using the actual standard OSS permissions or OSS ACL permissions (if present) on the file.

For more information about OSS NFS file system security, see the *Overview of NFS for Open System Services* and the *Open System Services NFS Management and Operations Guide*.

When using NFS with OSS filesets with objects protected by optional ACL entries, consider the following:

- NFS client/server interactions work most efficiently for read-only OSS filesets when the OSS filesets are mounted read-only on the NFS client systems instead of setting the readonly attribute in either the OSS NFS server configuration or OSS fileset configuration. NFS client attempts to write to a read-only OSS fileset are reported immediately to the NFS client application.

- If an OSS fileset has objects protected by optional OSS ACL entries, if you mount that fileset from NFS client systems as read-write, you must use mount options that disable write buffering. Because of the behavior of some NFS V2 clients, if you do not disable write buffering, the server might not receive the correct user ID information from the NFS client, which can result in write requests being denied or data being written to a file by a client that should have been denied write access. See the description of the OSS fileset NFSPERMMAP attribute in the *Open System Services NFS Management and Operations Guide*
- Changing the NFSPERMMAP attribute on an OSS fileset in which NFS clients currently have open files can confuse some NFS client software. See the discussion about changing the operating parameters of a fileset in the *Open System Services Management and Operations Guide*.

## HEADERS

### sys/acl.h

The **sys/acl.h** header file defines the following constants to govern the number of entries per ACL:

**NACLENTRIES**
The maximum number of entries per ACL, including base entries

**NACLBASE** The number of base entries

For compatibility with HP-UX, the variable name **NACLVENTRIES** is provided as an alias for **NACLENTRIES**.

The ACL structure **struct acl** is also defined and includes these fields:

**int** *a_type*; **/* type of entry */**
**uid_t** *a_id*; **/* group ID */**
**unsigned short** *a_perm*; **/* see <unistd.h> */**

The **sys/acl.h** header defines the set of valid values for the *a_type* field in addition to the valid values for the *cmd* parameter of the **acl( )** function.

## EXAMPLE PROGRAM

This program provides simple examples of **acl(2)** and **aclsort(3)** usage.

```
/* This program provides simple examples of acl(2) and aclsort(3) usage.
 * It adds a GROUP ACL entry (with read permissions) to the ACL of the
 * file.  The file pathname and group ID number are passed as command
 * arguments.
 * To run:
 *   addACLgroup <pathname> <group ID number>
 * This program performs the following steps:
 *   1. Acquires the count of ACL entries in the ACL on the file
 *      using acl(ACL_CNT).
 *   2. Allocates memory for the ACL buffer using malloc().
 *   3. Acquires the existing ACL on the file using acl(ACL_GET).
 *   4. Adds a new GROUP ACL entry to the end of the ACL buffer.
 *   5. Calls aclsort() to sort the ACL entries in the ACL buffer
 *      into the proper order.
 *   6. Sets the new ACL on the file using acl(ACL_SET).
```

```
 * If you run this program twice on the same file, it will report
 * an error in aclsort() as you are trying to add a second group ACL entry
 * for the same group id. aclsort() points to the ACL entry in error.
 */
#include <stdlib.h>
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <acl.h>
#include <errno.h>
#define READPERM 4
#define CALCCLASS 1

typedef struct acl  acl_t;
void printAcl( char *header, acl_t *aclEnt, int count )
{
  int i;
  printf("%s\n",header);
  for (i= 0; i < count; i++) {
     printf("acl entry %d ", i);
     printf("\ta_type = %d ", aclEnt[i].a_type );
     printf("\ta_id = %d ",   aclEnt[i].a_id );
     printf("\ta_perm = %o\n", aclEnt[i].a_perm );
  }
}
main( int argc, char *argv[])
{
  acl_t *aclEnt = 0;            /* pointer to ACL buffer */
  char  *pathname = 0;          /* pointer to pathanme */
  int prevCount, newCount;      /* counts of ACL entries */
  int groupId;                  /* group ID number for new ACL entry */
  int error;                    /* error variable */

  pathname = argv[1];           /* get ptr to pathname command argument */
  groupId = atoi(argv[2]);      /* get groupId command argument value */
  printf("Input pathname = %s, input groupId = %d\n", pathname, groupId);

  /* find out how many ACL entries in the existing ACL on the object */
  if (( prevCount = acl(pathname, ACL_CNT, NACLENTRIES, aclEnt)) == -1 ) {
     printf("acl(ACL_CNT) error= %d, text = %s\n", errno, strerror(errno));
     return 1;
  }
  printf("Number of ACL entries = %d\n", prevCount);
  /* Allocate space, reserving 1 extra ACL entry for the new GROUP entry */
  newCount = prevCount + 1;
  if (( aclEnt = (acl_t *) malloc( newCount * sizeof(acl_t))) == 0 ) {
     printf("malloc error= %d, text = %s\n", errno, strerror(errno));
     return 1;
  }
  /* Acquire the existing ACL on the object */
  if ((prevCount = acl(pathname, ACL_GET, prevCount, aclEnt)) == -1 ) {
     printf("acl(ACL_GET) error= %d, text = %s\n", errno, strerror(errno));
     free(aclEnt);
```

```
    return 1;
   }
  printAcl("Existing ACL entries", aclEnt, prevCount);
/* add new GROUP acl entry at the end of the ACL */
  aclEnt[ newCount-1 ].a_type = GROUP;
  aclEnt[ newCount-1 ].a_id = groupId;
  aclEnt[ newCount-1 ].a_perm = READPERM;
  printAcl("New ACL entries before aclsort()",aclEnt, newCount);

/* sort all of the ACL entries into proper order for acl( ACL_SET) */
  if ((error = aclsort(newCount, CALCCLASS, aclEnt)) != 0 ) {
     printf("aclsort() error = %d\n", error);
     free(aclEnt);
     return 1;
  }
  printAcl("New ACL entries after aclsort()", aclEnt, newCount);

  /* now set the new ACL on the object */
  if ((error = acl(pathname, ACL_SET, newCount, aclEnt)) == -1 ) {
     printf("acl(ACL_SET) error= %d, text = %s\n", errno, strerror(errno));
     free(aclEnt);
     return 1;
   }
  free( aclEnt );
  return 0;
}
```

## WARNINGS

You cannot use ACLs to restrict the access of the super ID.

Of the various file archive utilities (such as **ar**, Backup and Restore 2, **cpio**, **pax**, and **tar**), only the Backup and Restore 2 utility can back up and restore any optional ACL entries associated with an OSS file. For more information, see the ACL restrictions in the reference pages for the other file archive utilities.

## FILES

**sys/acl.h** Header file that supports the **acl( )** function.

**sys/aclv.h** Header file that includes the **sys/acl.h** header file for compatibility with HP-UX.

## RELATED INFORMATION

Commands: **chmod(1)**, **cp(1)**, **find(1)**, **getacl(1)**, **ln(1)**, **ls(1)**, **mv(1)**, **rm(1)**, **setacl(1)**, **fsck(1)**.

Functions: **access(2)**, **acl(2)**, **chmod(2)**, **chown(2)**, **creat(2)**, **mknod(2)**, **open(2)**, **stat(2)**, **aclsort(3)**.

## NAME

**ascii -** Describes the octal, hexadecimal, and decimal ASCII character sets

## DESCRIPTION

The octal character set is as follows:

Table 12−1. ASCII Character Set Octal Values

```
000 nul  001 soh  002 stx  003 etx  004 eot  005 enq  006 ack
007 bel  010 bs   011 ht   012 nl   013 vt   014 np   015 cr
016 so   017 si   020 dle  021 dc1  022 dc2  023 dc3  024 dc4
025 nak  026 syn  027 etb  030 can  031 em   032 sub  033 esc
034 fs   035 gs   036 rs   037 us   040 sp   041  !   042  "
043  #   044  $   045  %   046  &   047  ’   050  (   051  )
052  *   053  +   054  ,   055  -   056  .   057  /   060  0
061  1   062  2   063  3   064  4   065  5   066  6   067  7
070  8   071  9   072  :   073  ;   074  <   075  =   076  >
077  ?   100  @   101  A   102  B   103  C   104  D   105  E
106  F   107  G   110  H   111  I   112  J   113  K   114  L
115  M   116  N   117  O   120  P   121  Q   122  R   123  S
124  T   125  U   126  V   127  W   130  X   131  Y   132  Z
133  [   134  \   135  ]   136  ˆ   137  _   140  ‘   141  a
142  b   143  c   144  d   145  e   146  f   147  g   150  h
151  i   152  j   153  k   154  l   155  m   156  n   157  o
160  p   161  q   162  r   163  s   164  t   165  u   166  v
167  w   170  x   171  y   172  z   173  {   174  |   175  }
176  ˜   177 del
```

The hexadecimal character set is as follows:

Table 12−2. ASCII Character Set Hexadecimal Values

```
00 nul   01 soh   02 stx   03 etx   04 eot   05 enq   06 ack
07 bel   08 bs    09 ht    0a nl    0b vt    0c np    0d cr
0e so    0f si    10 dle   11 dc1   12 dc2   13 dc3   14 dc4
15 nak   16 syn   17 etb   18 can   19 em    1a sub   1b esc
1c fs    1d gs    1e rs    1f us    20 sp    21  !    22  "
23  #    24  $    25  %    26  &    27  ’    28  (    29  )
2a  *    2b  +    2c  ,    2d  -    2e  .    2f  /    30  0
31  1    32  2    33  3    34  4    35  5    36  6    37  7
3f  ?    38  8    39  9    3a  :    3b  ;    3c  <    3d  =
3e  >    40  @    41  A    42  B    43  C    44  D    45  E
46  F    47  G    48  H    49  I    4a  J    4b  K    4c  L
4d  M    4e  N    4f  O    50  P    51  Q    52  R    53  S
54  T    55  U    56  V    57  W    58  X    59  Y    5a  Z
5b  [    5c  \    5d  ]    5e  ˆ    5f  _    60  `    61  a
62  b    63  c    64  d    65  e    66  f    67  g    68  h
69  i    6a  j    6b  k    6c  l    6d  m    6e  n    6f  o
70  p    71  q    72  r    73  s    74  t    75  u    76  v
77  w    78  x    79  y    7a  z    7b  {    7c  |    7d  }
7e  ˜    7f del
```

The decimal character set is as follows:

Table 12–3. ASCII Character Set Decimal Values

<table>
<tr><td>0 nul</td><td>1 soh</td><td>2 stx</td><td>3 etx</td><td>4 eot</td><td>5 enq</td><td>6 ack</td></tr>
<tr><td>7 bel</td><td>8 bs</td><td>9 ht</td><td>10 nl</td><td>11 vt</td><td>12 np</td><td>13 cr</td></tr>
<tr><td>14 so</td><td>15 si</td><td>16 dle</td><td>17 dc1</td><td>18 dc2</td><td>19 dc3</td><td>20 dc4</td></tr>
<tr><td>21 nak</td><td>22 syn</td><td>23 etb</td><td>24 can</td><td>25 em</td><td>26 sub</td><td>27 esc</td></tr>
<tr><td>28 fs</td><td>29 gs</td><td>30 rs</td><td>31 us</td><td>32 sp</td><td>33 !</td><td>34 "</td></tr>
<tr><td>35 #</td><td>36 $</td><td>37 %</td><td>38 &</td><td>39 ’</td><td>40 (</td><td>41 )</td></tr>
<tr><td>42 *</td><td>43 +</td><td>44 ,</td><td>45 -</td><td>46 .</td><td>47 /</td><td>48 0</td></tr>
<tr><td>49 1</td><td>50 2</td><td>51 3</td><td>52 4</td><td>53 5</td><td>54 6</td><td>55 7</td></tr>
<tr><td>56 8</td><td>57 9</td><td>58 :</td><td>59 ;</td><td>60 <</td><td>61 =</td><td>62 ></td></tr>
<tr><td>63 ?</td><td>64 @</td><td>65 A</td><td>66 B</td><td>67 C</td><td>68 D</td><td>69 E</td></tr>
<tr><td>70 F</td><td>71 G</td><td>72 H</td><td>73 I</td><td>74 J</td><td>75 K</td><td>76 L</td></tr>
<tr><td>77 M</td><td>78 N</td><td>79 O</td><td>80 P</td><td>81 Q</td><td>82 R</td><td>83 S</td></tr>
<tr><td>84 T</td><td>85 U</td><td>86 V</td><td>87 W</td><td>88 X</td><td>89 Y</td><td>90 Z</td></tr>
<tr><td>91 [</td><td>92 \</td><td>93 ]</td><td>94 ˆ</td><td>95 _</td><td>96 ‘</td><td>97 a</td></tr>
<tr><td>98 b</td><td>99 c</td><td>100 d</td><td>101 e</td><td>102 f</td><td>103 g</td><td>104 h</td></tr>
<tr><td>105 i</td><td>106 j</td><td>107 k</td><td>108 l</td><td>109 m</td><td>110 n</td><td>111 o</td></tr>
<tr><td>112 p</td><td>113 q</td><td>114 r</td><td>115 s</td><td>116 t</td><td>117 u</td><td>118 v</td></tr>
<tr><td>119 w</td><td>120 x</td><td>121 y</td><td>122 z</td><td>123 {</td><td>124 |</td><td>125 }</td></tr>
<tr><td>126 ˜</td><td>127 del</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

## NAME

**environ -** Contains the user environment

## SYNOPSIS

**extern char **environ;**

## DESCRIPTION

An array of strings called the environment is made available by the **execl( )**, **execle( )**, **execlp( )**, **execv( )**, **execve( )**, **execvp( )**, **tdm_execve( )**, or **tdm_execvep( )** function when a process begins. The same array is optionally made available by the **tdm_spawn( )** or **tdm_spawnp( )** function when a process begins.

COBOL programs also have access to the environment when the COBOL SAVE ALL directive is used at compile time and the Guardian PARAM SAVE-ENVIRONMENT ON is used before starting an OSS shell to run the program.

By convention, these strings have the form *name=value*. The names used by various commands and utilities are:

**AS1** Specifies the pathname of the C or C++ compiler component used when binary assembly-code conversion to object code is requested. By default, the program **as1** in the directory **/usr/lib** is used. This environment variable is used for TNS/R-targeted compilations only.

**CCOMBE** Determines the pathname of the **ccombe** component of the C and C++ compilers. | **/usr/cmplr/ccombe** is the default location for the OSS environment. |

This environment variable is used for TNS/E-targeted compilations only.

**CACHE_CDS_SERVER**

Specifies the name of the CDS server to cache. The cached server is not required to be the initial CDS server. Used during CDS client configuration by the **dce_config** command.

**CACHE_CDS_SERVER_IP**

Specifies the IP address of the CDS server to cache; used by the **dce_config** command.

**CDPATH** Specifies the search path used for the **cd** command.

**CDS_ADVERTISEMENTS**

Controls the behavior of the CDS advertiser. When this variable has the value **n**, the CDS advertiser is started with the **-s** switch by the **dce_config** command, meaning the server does not send or receive advertisements.

The default is **y**.

**CDSD_DATABASE_DIR**

Specifies the location of the CDS database files, which is a Guardian subvolume holding NonStop SQL/MP tables. This value is a Guardian subvolume name, expressed in OSS pathname format and surrounded by quotation marks; for example, **"/G/volume/subvol"**.

Used by the **dce_config** command.

**CELL_ADMIN**
Specifies the principal name of the initial privileged user of the registry database (known as the "registry creator"). Used by the **dce_config** command during security server configuration.

**CELL_ADMIN_PW**
Specifies the default password assigned to the accounts created when the registry database is created, including the account for the registry creator. Used by the **dce_config** command.

The default is **-dce-**.

**CELL_NAME**
Specifies the name of the cell (without the **/.../**) on which the configuration is being performed. Used during security server configuration by the **dce_config** command.

**CFE** Specifies the pathname of the C or C++ compiler used when C or C++ source statements are present. By default, the program **cfe** in the directory **/usr/lib** is used.

This environment variable is used for TNS/R-targeted compilations only.

**check_time** Specifies whether to check client and server clock synchronization. (All lowercase characters is correct.)

Valid values are:

**y** Indicates time is checked

**n** Indicates time is not checked

The default is **y**.

Used by the **dce_config** command.

**CLONE_FROM**
Specifies the name of the virtual host to be used when cloning is performed. Used by the **dce_config** command.

This variable is ignored unless **CLONING_REQUIRED** is set to **y**. If **CLONING_REQUIRED** is set to **y**, **CLONE_FROM** must specify the name of a virtual host that is already installed.

**CLONING_REQUIRED**
Specifies whether binary files of another virtual host should be shared (cloned). Used by the **dce_config** command.

Valid values are:

**y** Indicates that cloning should occur

**n** Indicates that cloning should not occur

The default is **n**.

**COMP_ROOT**
Specifies a pathname prefix to be used to find the components of the **c89** utility.

**COPY_CONFIG_HOST**
Specifies the name of the virtual host to be used when copying for replica servers is performed. Used by the **dce_config** command.

This variable is ignored unless **COPY_CONFIG_INFO** is set to **y**. If **COPY_CONFIG_INFO** is set to **y**, then **COPY_CONFIG_HOST** must specify the name of a virtual host that is already installed.

**COPY_CONFIG_INFO**
Specifies whether the configuration should be copied from another virtual host. Used by the **dce_config** command.

Copying implies that an additional CDS or security server is being configured to be a replica of the virtual host named by the **COPY_CONFIG_HOST** environment variable.

Valid values are:

**y** Indicates that copying should occur

**n** Indicates that copying should not occur

The default is **n**.

**CPU_LIST** Specifies the processors to be used by the virtual host being configured. Used by the **dce_config** command.

Processor numbers must be separated by one or more spaces, and the list of numbers must be enclosed in quotation marks. If a specified processor is down or not available, the system allocates a replacement processor.

**CRON_NAMED**
Specifies the process name to be used when the **cron** utility is run.

Valid values must conform to Guardian process name rules, cannot begin with a Z, and must be specified in OSS pathname form (**/G/***process_name*) without the **$** character.

**DATEMSK** Specifies the full pathname for the file of date templates used with the **getdate( )** function.

**DCE_PRIVUSER**
Specifies the NonStop operating system user ID permitted to perform privileged operations, such as configuring servers using **dce_config**. This user ID must be a member of the super group.

The default is the super ID (255,255 with a scalar view of 65535).

**DCE_PROCESS_PREFIX**
Specifies one alphabetic character to be used as the prefix for virtual host process names. All processes started for a virtual host use this prefix.

The default is **Z**.

For example, DCE processes started when the default value is used have the Guardian process names $ZDCED, $ZSECD, and so forth.

Used by the **dce_config** command.

**DCE_SCP_PROCESS_NAME**

Specifies the Guardian process name of the Subsystem Control Point (SCP) process to be contacted by all DCE processes in the virtual host. The default is $ZNET.

If this variable is assigned a value, the **$** character in the SCP process name must be preceded by the shell \ character, as in:

**export DCE_SCP_PROCESS_NAME=\$ZNET1**

If the specified process is not responding or not running, **dce_config** uses the default of $ZNET. However, DCE demons and other processes either do not start or do not respond until the specified process is running.

**DCE_SOCKET_REUSE**

Specifies whether the IP address for the **dced** process is reused when the process is restarted. Valid values are:

| | |
|---|---|
| 0 (zero) | The address is not reused. |
| 1 | The socket **SO_REUSEADDR** option is used at the time of binding port 135 so that even if the port is used by another process, **dced** restarts sucessfully. |

The default is 0 (zero).

Used by the **dce_config** command.

**DCED_ADMIN**

Specifies whether the administrative group **dced-admin** should have permission to access and modify the access control lists that protect **dced** objects. Used by the **dce_config** command.

The value **y** allows the administrative group to access and modify local dced objects. If you use the value **y**, a privileged network user such as **cell_admin** is allowed local privileged access to the machine.

The value **n** restricts access and modification permission to the local host principal. If you use the value **n**, security is greater, but remote **dced** management is severely restricted.

The default is **y**.

**DCEVH**

Specifies the TCP/IP host name assigned to the virtual host for which operations are currently being performed. Used by the **dce_config** command, DCE demons, application servers, and clients.

The value specified for **DCEVH** is used when **TCPIP_PROCESS_NAME** is not specified.

If neither **DCEVH** nor **TCPIP_PROCESS_NAME** is specified, the default value of **DCEVH** is the name of the virtual host containing the **/opt/dcelocal** directory. If the **/opt/dcelocal** directory is not available, a default process name of **/G/ZTC0** is assumed, and the default value of **DCEVH** is the hostname attribute of **/G/ZTC0**.

If **TCPIP_PROCESS_NAME** and **DCEVH** are both specified, the value specified for **DCEVH** is ignored, and the hostname attribute of the specified process is used.

**DIR_REPLICATE**

Controls the replication of CDS directories when an additional CDS server is being created at DCE configuration time. Used by the **dce_config** command.

Valid values are:

**y** Causes **dce_config** to prompt for more directories to replicate

**n** Suppresses further replication

The default is **n**.

**DISPLAY_THRESHOLD**

Specifies the messages to write to the standard output file.

Valid values are:

**DEBUG**
**DETAIL**
**ERROR**
**WARNING**
**SUMMARY**
**VERBOSE**

The default is **SUMMARY**.

Used by the **dce_config** command.

**DO_CHECKS** Controls whether the prompt

```
Press <RETURN> to continue, CTRL-C to exit:
```

is returned when **dce_config** encounters a nonfatal error. This prompt forces the user to acknowledge the error and offers a way to exit **dce_config**.

Valid values are:

**y** Displays the prompt

**n** Does not display the prompt

The default is **y**.

When **DO_CHECK** has the value **y** during configuration of a security server and a security client, a check is made that a **kerberos5** entry exists in **/G/system/ztcpip/services**.

**DTS_CONFIG**

Specifies the type of configuration needed for a distributed time service (DTS) server during DCE client configuration. Used by the **dce_config** command.

Valid values are **clerk**, **global**, **local**, and **none**.

The default is **clerk**.

**ECOBFE** Determines the pathname of the **ecobol** compiler. **/G/system/system/ecobfe** is the default.

**EDITOR** Specifies the inline editor used by the shell. If the value of this variable ends in "vi" and the **VISUAL** variable is not assinged a variable, the corresponding inline editor option is enabled.

**ELD** Specifies the pathname of the TNS/E linker for PIC code used by the compiler utilities to link object and library files into an executable program or dynamic-link library when linking is requested. By default, the program **eld** in the directory **/usr/bin** is used.

This environment variable is used for TNS/E-targeted compilations only.

**EMS_COLLECTOR**
Specifies alternate collector processes to the **syslog( )** function.

**ENV** Specifies the path used to find the script to be executed when the shell is invoked.

**EXINIT** Provides a start-up list of commands read by the **vi** utility.

**EXIT_ON_ERROR**
Indicates whether **dce_config** exits in the event of a fatal error.

Valid values are:

**y** Indicates that **dce_config** exits when it encounters a fatal error

**n** Indicates that **dce_config** does not exit when it encounters a fatal error

The default is **n**.

This variable can help prevent a "here" file from getting out of sync with **dce_config**.

**FCEDIT** Specifies the default editor used for the **fc** command.

**FPATH** Specifies the search path used for shell function definitions.

Guardian PARAMs
Specify the names and values of Guardian environment PARAMs, as known to the **osh** command. The names and values are converted from the Guardian environment PARAM names and values. See the **osh(1)** reference page for details.

**HISTFILE** Specifies the pathname of the file used by the shell to store the command history.

**HISTSIZE** Specifies the number of previously entered commands accessible to the shell.

**HOME** Provides a user's login directory.

**HOST_NAME_IP**
Specifies the IP address of the virtual host on which **dce_config** is running.

**IFS** Specifies the internal field separators used in shell scripts.

**JAVA_HOME** Specifies the pathname for the most current installed version of the NonStop Java Server environment.

**KEYSEED** Specifies the character string used to seed the random key generator to create the master key for the master database and each slave database. Each database has its own master key and keyseed. Used in security server configuration by the **dce_config** command.

**LANG** Sets the locale to be used for all categories, unless overridden by **LC_ALL**, **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC**, or **LC_TIME** environment variables. The **LANG** and **LC_*** environment variables can each have one of these values:

**C** For the C locale

**POSIX** For the POSIX locale

*ll_TT.CODESET*

*ll* Is a 2-letter, lowercase abbreviation for the language name. The abbreviations come from ISO 639. For example:

| | |
|---|---|
| **en** | English |
| **fr** | French |
| **ja** | Japanese |
| **de** | German (from Deutsch) |

*TT* Is a 2-letter, uppercase abbreviation for the territory name. The abbreviations come from ISO 3166. For example:

| | |
|---|---|
| **US** | United States of America |
| **JP** | Japan |
| **NL** | The Netherlands |
| **ES** | Spain (from España) |

*CODESET* Is the name of the code set or encoding method. For example:

| | |
|---|---|
| **ASCII** | ASCII |
| **ISO8859-1** | ISO 8859-1 |
| **AJEC** | Japanese EUC |

Some examples of full locale names are:

| | |
|---|---|
| **en_US.ISO8859-1** | English, United States of America, ISO 8859-1 |
| **fr_FR.ISO8859-1** | French, France, ISO 8859-1 |
| **fr_CH.ISO8859-1** | French, Switzerland, ISO 8859-1 |
| **ja_JP.AJEC** | Japanese, Japan, EUC |

**LAN_NAME** For a multiple-LAN configuration, specifies the internal name of the LAN (in the LAN profile). Used in CDS server configuration by the **dce_config** command.

**LC_ALL** Sets the locale for all categories and overrides any other locale environment variables set. See the description of **LANG** for locale name syntax.

**LC_COLLATE**
Sets the locale to be used for collating strings. See the description of **LANG** for locale name syntax.

**LC_CTYPE** Sets the locale to be used for classifying characters. See the description of **LANG** for locale name syntax.

**LC_MESSAGES**
Sets the locale to be used for displaying messages. See the description of **LANG** for locale name syntax.

**LC_MONETARY**
Sets the locale to be used for formatting monetary values. See the description of **LANG** for locale name syntax.

**LC_NUMERIC**
Sets the locale to be used for formatting and parsing numeric values. See the description of **LANG** for locale name syntax.

**LC_TIME** Sets the locale to be used for formatting and parsing date and time values. See the description of **LANG** for locale name syntax.

**LD** Specifies the pathname of the TNS/R linker for PIC code used by the compiler utilities to link object and library files into an executable program or dynamic-link library when linking is requested. By default, the program **ld** in the directory **/usr/bin** is used. This environment variable is used for TNS/R-targeted compilations only.

**LOCPATH** Specifies the sequence of directories, separated by colons, to be searched by the **iconv_open( )** function when looking for the table-driven **iconv** converter modules.

**LOG_THRESHOLD**
Specifies the minimum priority log messages to write to the log file, **/tmp/dce_config.log**. Used by the **dce_config** command.

Valid values are:

**DEBUG**
**DETAIL**
**ERROR**
**WARNING**
**SUMMARY**
**VERBOSE**

The default is **VERBOSE**.

**LOGNAME** Specifies the user's login name, as known to the **osh** command. The value is converted from the Guardian PARAM LOGNAME.

**MAKEFLAGS**
Lists the environment variables for the **make** utility to process. Setting a variable in **MAKEFLAGS** overrides the setting of that variable in the shell.

**MANPATH** Sets the path used by the **man** command to look for files to display. The default pathname is **/usr/share/man**.

**MSGVERB** Defines which message components are sent by the **fmtmsg( )** function to the standard error file.

**MULTIPLE_LAN**
Indicates whether to configure the node with multiple LAN capabilities.

Valid values are:

**y** Indicates configure with multiple LAN capabilities

**n** Indicates do not configure with multiple LAN capabilities

The default is **n**.

Used in CDS configuration by the **dce_config** command.

**MXCMP** Determines the pathname of the NonStop SQL/MX release 1 compiler. **/G/system/system/mxcmp** is the default.

Used by the compiler utilities.

**MXCMPUM** Determines the pathname of the NonStop SQL/MX release 2 compiler. **/usr/tandem/sqlmx/bin/mxCompileUserModule** is the default.

Used by the compiler utilities.

**MXSQLC** Determines the pathname of the NonStop SQL/MX preprocessor, **mxsqlc**. **/usr/tandem/sqlmx/bin/mxsqlc** is the default.

Used by the **c89** command.

**MXSQLCO** Determines the pathname of the NonStop SQL/MX preprocessor, **mxsqlco**. **/usr/tandem/sqlmx/bin/mxsqlco** is the default.

Used by the **nmcobol** command.

**NLD** Specifies the pathname of the non-PIC TNS/R linker used by the compiler utilities to link object and library files into an executable program or shared run-time library when linking is requested. By default, the program **nld** in the directory **/usr/bin** is used.

This environment variable is used for TNS/R-targeted compilations only.

**NLSPATH** Specifies the sequence of directories, separated by colons, to be searched by the **catopen( )** function when looking for message catalogs. The meanings of the variables in the **NLSPATH** environment variable are:

**%N** The value passed in the *name* parameter of **catopen( )**.

**%L** The current locale name defined for the **LC_MESSAGES** category: for example, **fr_BE.ISO8859-1**.

**%l** The language element from the current locale name: for example, **fr**.

**%t** The territory element from the current locale name: for example, **BE**.

**%c** The code-set element from the current locale name: for example, **ISO8859-1**.

**%%** A single % (percent sign) character.

**PATH** Specifies the sequence of directories, separated by colons, to be searched by the **sh** utility, the **system** command, the **execvp( )** function, and so forth, when looking for an executable file. The **osh** command can convert the Guardian PARAM PATH to this value.

**PMSEARCHLIST**
Specifies the values used by the **gtacl** command to resolve a Guardian file identifier.

**PRINTER** Specifies the name of the default printer.

**PS1** Specifies the primary prompt string used by the shell.

**PS2** Specifies the secondary prompt string used by the shell.

**PS3** Specifies the selection prompt string used by the shell within a loop.

**PS4** Specifies the prompt string used by the shell during an execution trace.

**PWD** Specifies the user's initial working directory, as known to the **osh** command. The value is converted from the Guardian PARAM PWD.

**PWD_MGMT_SVR**
Specifies the pathname of the Password Management server. The default value is **/opt/dcelocal/bin/pwd_strengthd**. Used in Password Management server configuration by the **dce_config** command.

**PWD_MGMT_SVR_OPTIONS**
Specifies the default options for the Password Management server (**pwd_strengthd**). The value of the variable is set to **-v** (verbose) at server configuration.

Used by the **dce_config** command.

**REMOVE_PREV_CONFIG**
Indicates whether to remove all remnants of previous configurations before performing the new configuration.

Valid values are:

**y** Indicates remove all remnants

**n** Indicates do not remove all remnants

If you set this variable to **y**, **dce_config** removes all configured components each time you configure any component, and you must reconfigure them all.

Used in all component configurations by the **dce_config** command.

**REMOVE_PREV_INSTALL**

Indicates whether to remove all remnants of previous DCE installations before performing the new install.

Valid values are:

**y** Indicates remove all remnants

**n** Indicates do not remove all remnants

If you set this variable to **y**, **dce_config** automatically removes all installed components each time you install any component, and you must reinstall them all.

Used in all component installations by the **dce_config** command.

**REP_CLEARINGHOUSE**

Specifies the name for a new clearinghouse. Used in additional CDS server configuration by the **dce_config** command.

**REPLICATE_ALL_DIRS**

Specifies whether to replicate all directories from the master CDS server database to the additional CDS server database during additional CDS server configuration. Used by the **dce_config** command.

The value **y** indicates that all directories should be replicated.

The value **n** indicates that no directories should be replicated.

The default is **n**.

**REPLICATE_DIR_LIST**

Specifies a list of directories to be replicated. Used by the **dce_config** command.

Directory pathnames must be separated by one or more spaces, and the list of directories must be enclosed in quotation marks.

If this variable is not specified, the user is prompted for a directory list.

**_RLD_FIRST_LIB_PATH**

Specifies a list of directory pathnames to be searched by the **rld** loader before searching public libraries or locations specified by the linker. The list has a format similar to that of the **PATH** environment variable, with individual entries separated by colons (**:**).

For more information, see the **dlopen(3)** reference page.

**_RLD_LIB_PATH**

Specifies a list of directory pathnames to be searched by the **rld** loader before searching default locations. The list has a format similar to that of the **PATH** environment variable, with individual entries separated by colons (**:**).

For more information, see the **dlopen(3)** reference page.

**SEC_REPLICA**

Specifies the name of the security replica database. Used by the **dce_config** command.

The default value is the name of the host being configured.

**SEC_SERVER**
Specifies the name of the machine on which the cell's master security server runs. Used in security client configuration by the **dce_config** command.

**SHELL** Specifies the full pathname of the user's login shell.

**SOCKET_TRANSPORT_NAME**
Specifies the process name of the OSS sockets transport provider process to be used by the **inetd** process.

**SQLCFE** Specifies the pathname of the embedded NonStop SQL/MP preprocessor and compiler normally invoked by the **c89** utility. By default, the program **sqlcfe** in the directory **/usr/lib** is used.

This environment variable is used for TNS/R-targeted compilations only.

**SQLCOMP** Specifies the pathname of the final-stage NonStop SQL/MP compiler invoked by the **c89** utility when embedded SQL is present and the program file is not a shared resource library file. By default, the program **sqlcomp** in the directory **/G/system/system** is used.

**SQLMX_PREPROCESSOR_VERSION**
Indicates the preprocessor rules and features to be used. Specifying the value 800 causes rules and features associated with release 1.8 to be used; the **mxcmp** compiler is used and only MDF files and annotated source files are produced, while rules and features associated with release 2.0 and later are ignored. Specifying a value of 1200 or larger or not specifying a value causes rules and features associated with release 2.0 and later to be used; the **mxCompileUserModule** compiler is used and annotated source files that contain embedded module definitions are produced instead of MDF files, while restrictions associated with release 1.8 or earlier are ignored.

**SWAPVOL** Specifies the disk volume used for working files by Guardian processes created by the TNS **c89** utility. This variable must evaluate to a Guardian disk volume: for example, **/G/scratch** or $SCRATCH. By default, the user's Guardian default volume is used.

**SYNC_CLOCKS**
Indicates whether to synchronize all client clocks with the security server clock. Used by the **dce_config** command.

Valid values are:

**y** Indicates that client and server clocks will be synchronized

**n** Indicates that client and server clocks will not be synchronized

If this variable is set to **n** and if clocks are out of synchronization by more than the value specified in the **TOLERANCE_SEC** variable, the user is prompted to synchronize them. This variable is valid only if the **CHECK_TIME** variable is set to **y**.

**TANDEM_ALT_SRL**

Controls whether the shared resource library is placed in an alternate location. This value is a Guardian filename expressed in OSS pathname format and surrounded by quotation marks: for example, **"/G/volume/subvol/ldce"**. The default is "", meaning that the shared resource library is placed in **/G/system/zdce/ldce**.

Used by the **dce_config** command.

**TANDEM_INSTALL_DIR**

Specifies the location of the **pax** files for installation. This value is a Guardian subvolume name expressed in OSS pathname format and surrounded by quotation marks: for example, **"/G/isv/zdce"**.

Used by the **dce_config** command.

**TCPIP_PROCESS_NAME**

Specifies the Guardian process name for the TCP/IP stack of the virtual host. Used by the **dce_config** command.

If neither **DCEVH** nor **TCPIP_PROCESS_NAME** is specified, the default value of **DCEVH** is the name of the virtual host containing the **/opt/dcelocal** directory. If the **/opt/dcelocal** directory is not available, a default process name of **/G/ZTC0** is assumed, and the default value of **DCEVH** is the hostname attribute of **/G/ZTC0**.

If **TCPIP_PROCESS_NAME** and **DCEVH** are both specified, the value specified for **DCEVH** is ignored, and the hostname attribute of the specified process is used.

**TCPIP_RESOLVER_NAME**

Specifies the OSS pathname to be used instead of **/etc/resolv.conf** to identify the dynamic name server to be used when resolving Internet addresses. Equivalent to the Guardian environment DEFINE =TCPIP^RESOLVER^NAME.

**TCPIP_RESOLVER_ORDER**

Controls the search order for TCP/IPv6 when OSS socket calls require access to addresses for a given host. The **/etc/ipnodes** and **/etc/hosts** files are searched as follows by default:

— If neither file exists, the domain name server (DNS) is checked for the host information.

— For an IPv4 host address, **/etc/ipnodes** is checked; if the host is not found, **/etc/hosts** is checked.

— For an IPv6 address, only **/etc/ipnodes** is checked.

For an IPv4 address, if **/etc/hosts** does not exist, the DNS is checked last.

When **/etc/hosts** exists, the values declared for the **TCPIP_RESOLVER_ORDER** environment variable can be used to control the search as follows:

**DNSONLY** Only the DNS is checked.

**HOSTFILEONLY**
Only **/etc/hosts** is checked.

**DNS-HOSTFILE**
The DNS is checked first; if the host is not found, **/etc/hosts** is checked.

**HOSTFILE-DNS**
**/etc/hosts** is checked first; if the host is not found, the DNS is checked.

**TERM** Specifies the type of terminal for which output must be prepared. This information is used by commands, such as **vi** or **more**, that can exploit special terminal capabilities. (See the **termcap(4)** reference page for a list of terminal types.)

**TERMCAP** Specifies a string describing the terminal in the **TERM** environment variable or, if it begins with a **/** (slash), the name of the **termcap** file. (See **TERMPATH**.) This string applies only to programs using a **termcap** file (only for compatibility).

**TERMINFO** Points to the directory containing **terminfo** database files. The **tic** command uses the value of this variable.

**TERMPATH** Specifies a sequence of pathnames of **termcap** files, separated by colons or spaces, which are searched for terminal descriptions in the order listed. The default is:

```
$HOME/.termcap:/usr/share/lib/termcap
```

**TERMPATH** is ignored if **TERMCAP** contains a full pathname. This string applies only to programs using the **termcap** file (only for compatibility).

**TIME_SERVER**
Specifies the virtual host that the security client will try to synchronize its clock against. This host must have a DTS server (**dtsd**) running on it. The recommended choice for the host is the one running the master security server (the name specified in the **SEC_SERVER** variable).

Used by the **dce_config** command.

**TMOUT** Specifies the number of seconds the shell waits for a response to a prompt before timing out.

**TMPDIR** Specifies a pathname that overrides the default directory for temporary files, **/tmp**. (Used by the **c89** utility.)

**TOLERANCE_SEC**
Specifies the number of seconds a client system clock can differ from the security server system clock before either the user is prompted to synchronize the clocks or the clocks are synchronized automatically.

The default is 120 seconds.

Both the security service and the CDS service require that there be no more than a 5-minute difference between the clocks on any two nodes in a cell.

Used by the **dce_config** command.

**TOTAL_CLERKS**

Specifies the number of CDS clerks for this host. On NonStop DCE systems, CDS clerks are shared among users (unlike some other DCE systems, which use one CDS clerk for each user ID).

The default is 1.

Used by the **dce_config** command.

**TZ** Specifies the time zone used by the shell and by time functions

to override the default timezone. The value of TZ has the following form:

[**:**]*stdoffset*[*dst*[*offset*][**,***start*[/*time*],end[/*time*]]]

*std* and *dst* Indicates no less than three, nor more than **TZNAME_MAX**, bytes that are the designation for the standard (*std*) or the alternative (*dst*, such as Daylight Savings Time) timezone. Only *std* is required; if *dst* is omitted, then the alternative time does not apply in this locale. Uppercase and lowercase letters are allowed. Any graphic characters except a leading colon (**:**) or digits, the comma (**,**), the minus sign (**-**), the plus sign (**+**), and the null character can appear in these fields.

If preceded by a -, the timezone is east of the Prime Meridian; otherwise, the timezone is west of the Prime Meridian (a condition that can be indicated by an optional preceding +).

*offset* Indicates the value to add to or subtract from the local time to arrive at Coordinated Universal Time. The offset has the form:

*hh*[**:***mm*[**:***ss*]]

The hour (*hh*) is required and can be a single digit. The minutes (*mm*) and seconds (*ss*) are optional.

The *offset* following *std* is required. If no *offset* follows *dst*, the alternative time is assumed to be one hour ahead of standard time. One or more digits can be used; the value is always interpreted as a decimal number. The hour is between 0 (zero) and 24, and the minutes (and seconds) are between 0 (zero) and 59. Use of values outside these ranges causes undefined behavior.

*date*[/*time*]**,***date*[/*time*]

The rule that indicates when to change to and back from the alternative time, where the first date describes when the change from standard to alternative time occurs and the second date describes when the change back happens. Each time field describes when, in current local time, the change to the other time is made. The format of *date* is one of the following:

**J***n* The Julian day *n* in the range 1 through 365. Leap days are not counted. That is, in all years including leap years, February 28 is day 59 and March 1 is day 60. You cannot refer explicitly to February 29.

*n* The zero-based Julian day in the range 0 through 365. Leap days are counted, and you can refer to February 29.

**M***m***.***n***.***d* The *d*th day (in the range 0 through 6) of week *n* (in the range 1 through 5) of month *m* of the year (in the range 1 through 12). Week 1 is the first week in which the day occurs. Week 5 means "the last *d* day in month *m*, which might occur in either the fourth or the fifth week. Day zero is Sunday.

*time* has the same format as *offset* except that no leading sign (**-** or **+**) is allowed. The default, if *time* is omitted, is 02:00:00.

**UGEN** Specifies the pathname of the C or C++ compiler component used when binary assembly code is requested. By default, the program **ugen** in the directory **/usr/lib** is used.

This environment variable is used for TNS/R-targeted compilations only.

**UNCONFIG_HOST_PRESET**
Specifies the name of the virtual host to be unconfigured. Used with the **unconfigure** option by the **dce_config** command.

**UOPT** Specifies the pathname of the C or C++ compiler component used when optimization is requested. By default, the program **uopt** in the directory **/usr/lib** is used.

This environment variable is used for TNS/R-targeted compilations only.

**UPDATE_ALL_CLONES**
Specifies whether the configurations of all existing clones of the current virtual host should be updated with the files being installed.

Valid values are:

**y** Indicates that clones should be updated

**n** Indicates that clones should not be updated

The default is **n**.

Used by the **dce_config** command.

**USE_DEF_MSG_PATH**
Specifies whether to use the default pathname when installing DCE message catalogs. Used by the **dce_config** command.

The value **y** indicates that message catalogs should be installed in the default directory **/usr/lib/nls/msg/en_US.ISO8859-1**.

The value **n** indicates that the user should be prompted to enter a directory pathname.

The default is **y**.

**USER** Specifies the login name of the user.

**UPDATE_DEFAULT_LIBDCESO**
Specifies whether the default **/usr/lib/libdce.so** file should be updated with the shared run-time library being installed.

Valid values are:

**y** Indicates that the update should occur

**n** Indicates that it should not occur

The default is **n**.

This variable can be used by the **dce_config** command only for TNS versions of NonStop DCE.

**UTILSGE** Specifies whether a shell utility attempts to include the **/E** or **/G** directories when recursively processing a pathname. This variable also can be tested by an application program to make the same determination.

Valid values are:

**NOE** Do not include the **/E** directory.

**NOG** Do not include the **/G** directory.

**NOG:NOE** Do not include either the **/E** or **/G** directory.

The default includes both the **/E** and **/G** directories.

**VISUAL** Specifies the inline editor used by the shell in visual mode.

**ZCPU** Specifies the processor number of the processor executing the process for which **ZCPU** was defined. This environment variable is set by the Kernel subsystem persistence manager for processes associated with process objects that are defined with the CPU ALL or CPU LIST attributes. If the variable is inherited by a process that has been spawned to another processor, the value might not be correct.

Additional names can be placed in the environment by the shell **export** command, by using *name=value* arguments. It is unwise to change the values of certain shell variables that are frequently exported by **.profile** files, such as **PS1**, **PS2**, and **IFS**.

## RELATED INFORMATION

Commands: **c89(1)**, **dce_config(8)** if installed, **gtacl(1)**, **osh(1)**, **sh(1)**.

Functions: **catopen(3)**, **exec(2)**, **getenv(3)**, **iconv_open(3)**, **putenv(3)**, **syslog(3)**, **tdm_execve(2)**, **tdm_execvep(2)**.

Files: **termcap(4)**.

## STANDARDS CONFORMANCE

HP extensions to the XPG4 Version 2 specification are:

- The **AS1**, **CCOMBE**, **CFE**, **COMP_ROOT**, **CRON_NAMED**, **EMS_COLLECTOR**, Guardian PARAMs, **JAVA_HOME**, **LOCPATH**, **NLD**, **PMSEARCHLIST**, **_RLD_FIRST_LIB_PATH**, **_RLD_LIB_PATH**, **SQLCFE**, **SQLCOMP**, **SWAPVOL**, **UGEN**, **UOPT**, **UTILSGE**, and **ZCPU** environment variables

- All environment variables used by the **dce_config** utility

## NAME

**errno -** Returns the error condition value

## SYNOPSIS

**#include <errno.h>**

## DESCRIPTION

The **errno** external variable contains the most recent error condition set by a function. The symbolic error names returned by a function and descriptions of each error condition are shown in the **ERRORS** section in the individual function reference pages. The **errno.h** header file contains a list of all symbolic error names and a one-line description of each.

The following is a list of the symbolic error names and the error condition each name describes:

[E2BIG] Argument list too long. The sum of the number of bytes used by the new process image's argument list and environment list is greater than the allowed system limits.

[EACCES] Permission denied. The program attempted to access a file in a way forbidden by its file access permissions.

[EADDRINUSE]
Address in use. The program tried to allocate an address that is already allocated.

[EADDRNOTAVAIL]
Can't assign requested address. The program tried to allocate an address that does not exist or cannot be allocated.

[EAFNOSUPPORT]
Address family not supported. The program requested an address in an address family not supported by the protocol family.

[EAGAIN] Resource temporarily unavailable. A system resource is temporarily unavailable, and later calls to the same routine might finish normally.

[EALREADY] Operation already in progress. The program attempted to begin an operation already in progress.

[EBADCF] C file not odd-unstructured. A C file (Guardian file code 180) is not an odd-unstructured file.

[EBADDATA] Invalid data in buffer. A message buffer contains invalid data.

[EBADF] Bad file descriptor. A file descriptor parameter is out of range or refers to no open file, or a read (write) request is made to a file that is open only for writing (reading).

[EBADFILE] File type not supported. A file access error occurred, or the file is of an unsupported type and cannot be opened.

[EBADMSG] An invalid message tag was found. There is no corresponding message for the message tag.

[EBADSYS] Invalid socket call. The program specified an unrecognized node name or node number in a socket call.

[EBIGDIR] The positioning within an OSS directory failed because there were more than 65535 file names beginning with the same two characters in the directory.

[EBUSY] Mount device busy. The program attempted to use a system resource that is not currently available, because it is being used by another process in a manner that would conflict with the request being made by this process.

[ECHILD] No child process. The **wait( )** or **waitpid( )** function was executed by a process that had no existing or unwaited-for child process.

[ECONNABORTED]
Software caused connection abort. Software on the connection path aborted the connection.

[ECONNREFUSED]
Connection refused. The other end of a requested connection refused to permit the connection to be made.

[ECONNRESET]
Connection reset by remote host. The connection was reset by the remote host.

[ECWDTOOLONG]
One of the following situations exists:

- The pathname of the current working directory is longer than the **PATH-MAX** value when the **getcwd( )** function was called.
- The length of the absolute pathname generated by the Guardian procedure call FILENAME_TO_PATHNAME_ is longer than **PATH_MAX**.
- The length of the absolute pathname generated by an internal Guardian procedure call is longer than **PATH_MAX**.

[EDEADLK] Deadlock condition. An attempt was made to lock a system resource that would have resulted in a deadlock situation.

[EDEFINEERR]
An error exists in a Guardian DEFINE.

[EDESTADDRREQ]
Destination request required. The program omitted a required destination address.

[EDOM] Argument out of range. A function parameter evaluates to a value that is out of range (too large or too small).

[EEXIST] File exists. An existing file was mentioned in an inappropriate context; for example, as a new link name in the **link( )** function.

[EFAULT] Bad address. The system detected an invalid address when attempting to use a parameter passed to a call.

[EFBIG] File too large. The size of a file would exceed the maximum file size of an implementation.

[EFILEBAD] Corrupt Guardian file or bad EDIT file structure. The program used the **open( )** or **read( )** function for an EDIT file (Guardian file code 101) in **/G** (the Guardian file system) that has an internal structure problem.

[EFSBAD] Fileset catalog internal consistency error. The program attempted an operation involving a fileset with a corrupted fileset catalog. This error will also be returned if there is a consistency error detected in an SMF catalog, between an SMF catalog and a disk process catalog, or an internal inconsistency error in an SMF files label. When creating or unlinking a file on an SMF Virtual Disk, this error is returned if the installed OSS Name Server and SMF Virtual Disk Process are incompatible or if the SMF Virtual Disk Process was unable to obtain volume status information from its associated SMF pool process.

[EFSERR] File system internal error. The program attempted an operation that failed because of a system programming error. Follow site-defined procedures for reporting software problems.

[EGUARDIANLOCKED]
Guardian record or file locked. The program used the **write( )** function for an object in **/G** (the Guardian file system) that has a record or file lock, resulting from a call to one of the following Guardian procedures:

LOCKFILE
LOCKREC
READLOCK
READLOCKX
READUPDATELOCK
READUPDATELOCKX

[EGUARDIANOPEN]
OSS **rename( )** or **unlink( )** function used on open Guardian file. An attempt was made to rename a file to an open Guardian file or to unlink a Guardian file opened with the Guardian FILE_OPEN_ procedure.

[EHAVEOOB] Out-of-band urgent data pending. Before receiving or sending normal data over a network connection, the program must read the out-of-band data by calling the **recv( )** function with the **MSG_OOB** flag set.

[EHLDSEM] A semaphore undo operation is occurring for an OSS process that has called a function in the **tdm_exec** or **tdm_spawn** set of functions to start a process in another processor.

[EHOSTDOWN]
Host is down. An access path has been broken or cannot be completed because a node has left the network.

[EHOSTUNREACH]
No route to host. No path exists to a node required by the process.

[EIDRM] Identifier removed. A required identifier has been removed.

[EILSEQ] Illegal byte sequence. An invalid wide character or a similarly invalid byte sequence has been detected.

[EINPROGRESS]
Operation in progress. A requested operation has begun.

[EINTR] Interrupted function call. An asynchronous signal was caught by the process during the execution of an interruptible function.

[EINVAL] Invalid function parameter. One of the following conditions exists:

- The program supplied an invalid parameter value.
- The system does not support execution of a new program file in the binary format used by a specified program file.

[EIO] I/O error. Some physical input or output error has occurred due to one of the following conditions:

- A file cannot be opened because of an input or output error.
- Data has been lost during an input or output transfer.
- A file cannot be accessed when creating or unlinking a file on an SMF logical volume and the SMF Virtual Disk Process encountered an error.
- The process is in a background process group and the controlling tty is either ignoring or blocking SIGTTIN, or the process group is orphaned.
- A Guardian error has occurred, such as error 66 (FEDEVDOWN) or error 201 (FEPATHDOWN) during a read or write.
- The OSS NS encountered unexpected read/write errors to fileset catalogs when communicating with other processes like DP2, pipe servers, and local socket servers.

[EISCONN] Socket is connected. The program attempted to use a socket that is already in use.

[EISDIR] Is a directory. The program attempted to open an OSS directory with **open( )** function write mode specified, or a directory in **/G** with any mode specified.

[EISGUARDIAN]
OSS operation on Guardian file descriptor. The program attempted an OSS operation involving a Guardian file descriptor.

[ELOOP] Too many symbolic links. The program found too many symbolic links during pathname resolution.

[EMFILE] Maximum number of files open. The program attempted to open more than the maximum number of file descriptors allowed in this process.

[EMLINK] Too many links. An attempt was made to have the link count of a single file exceed allowed system limits.

[EMSGQNOTRUNNING]
The OSS message-queue server for the requested message queue is not currently running.

[EMSGSIZE] Message too long. The specified message contained too many bytes.

[ENAMETOOLONG]
File name too long. One of the following is too long:

- A pathname specified in a function call
- A component of a pathname specified in a function call
- The intermediate result of pathname resolution when a symbolic link is part of a pathname specified in a function call

Use the **pathconf( )** function to obtain the applicable system limits.

[ENETDOWN]
Network down. The last path between the node and the network went down.

[ENETRESET] Network dropped connection on reset. The connection was dropped because the network was reset.

[ENETUNREACH]
Network unreachable. No path exists between the node and the network.

[ENFILE] File table overflow. Too many files are currently open in the system. The system has reached its predefined limit for simultaneously open files and temporarily cannot accept requests to open another one.

[ENOBUFS] Buffer space unavailable. No buffer space is available.

[ENOCPU] CPU unavailable. The program selected a processor that either does not exist, is down, or is unavailable for process creation.

[ENOCRE] Non-CRE process needs CRE-dependent service. The process is not compliant with the Common Run-Time Environment (CRE) but requested a service that depends on CRE.

[ENODATA] No data sent or received. No message or stream queue exists, or no data was sent or received.

[ENODEV] No such device. The program attempted to apply an inappropriate function to a device; for example, trying to read a write-only device, such as a printer.

[ENOENT] No such file, directory, or socket transport provider. A component of a specified pathname does not exist, the pathname is an empty string, a specified provider is no longer running, or a specified provider does not exist.

[ENOERR] No error occurred. This is the default value for **errno**.

[ENOEXEC] Executable program file format error. A request was made to execute a program file that, although it has the appropriate permissions, is not in the format required by the implementation for executable files.

[ENOIMEM] Insufficient internal memory. There is insufficient system code space in the processor to complete the operation.

[ENOLCK] No record locks available. A system-imposed limit on the number of simultaneous file and record locks has been reached, and no more are currently available.

[ENOMEM] Insufficient user memory. The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

[ENOMSG] No message. There is no message of the requested type.

[ENONSTOP] NonStop programming logic error exists. The program is written to use NonStop system features but has requested an operation incompatible with correct use of those features.

[ENOPROTOOPT]
Protocol not available. The requested protocol is not available.

[ENOREPLY] No reply in buffer. There is no reply in the message buffer.

[ENOROOT] Root fileset is not started. The program attempted an operation while the root fileset (fileset 0) was unavailable.

This error can occur after failure and restart of an OSS name server until the fileset has been repaired and remounted.

[ENOSPC] No space left on device. During the **write( )** function on a regular file or when extending a directory, there is no free space left on the device.

[ENOSYS] Function not implemented. An attempt was made to use a function that is not available in this implementation.

[ENOTCONN] Socket not connected. The socket is not connected.

[ENOTDIR] Not a directory. The program attempted a directory operation on an object that is not a directory.

[ENOTEMPTY]
Directory not empty. A directory with entries other than **.** (dot) and **..** (dot-dot) was supplied when an empty directory was expected.

[ENOTOSS] Not an OSS process. The program has called a function that can be called only from an OSS process.

[ENOTSOCK] Not a socket. The program attempted a socket operation on an object that is not a socket.

[ENOTSUP] Operation not supported on referenced object. The program attempted an operation that is not allowed on the referenced object.

[ENOTTY] Not a tty device. The program attempted a tty operation on an object that is not a tty device.

[ENXIO] No such device or address. An invalid device or address was specified during an input or output operation on a special file. One of the following events occurred:

- A device was specified that does not exist, or a request was made beyond the limits of the device.
- The fileset containing the requestor's current working directory or root directory is not mounted. This error can occur after failure and restart of an OSS name server until the fileset has been repaired and remounted.

[EOPNOTSUPP]
Operation not supported on sockets. The program attempted to perform an operation that is not valid on a socket.

[EOSSNOTRUNNING]
Open System Services is not running or not installed. The program attempted an operation on an object in the OSS environment while a required system process was not available.

[EOVERFLOW]
The program attempted to perform an operation on a file at a position beyond the offset maximum established when the file was opened.

[EPARTIAL] Partial buffer received. Only a partial buffer of message data was received.

[EPERM] One of the following conditions exist:

- Not owner, permission denied. An attempt was made to perform an operation limited to processes with appropriate privileges or limited to the owner of a file or other resource.
- The program attempted an operation on a SEEP-protected fileset. Valid for J06.15 and later J-series RVUs, and H06.26 and later H-series RVUs.

[EPFNOSUPPORT]
Protocol family not supported. The program specified a protocol family that is not supported.

[EPIPE] Broken pipe or no reader on socket. The program attempted to write on a pipe, FIFO, or socket for which there is no process to read the data.

[EPROTONOSUPPORT]
Protocol not supported. The program specified a protocol that is not supported.

[EPROTOTYPE]
Wrong protocol type. The program specified the wrong protocol for the type of socket.

[ERANGE] Value out of range. A program expression evaluated to a value that is out of range (too large or too small).

[EROFS] Read-only fileset. The program attempted to modify a file or directory on a fileset that is read only.

[ESHUTDOWN]
Can’t send after socket shutdown. The program attempted to send data after the socket shut down.

[ESOCKTNOSUPPORT]
Unsupported socket type. The program specified a socket type that is not supported.

[ESPIERR] SPI interface error. The Subsystem Programmatic Interface (SPI) used by an OSS component has returned an error indication.

[ESPIPE] Invalid seek. The program attempted to access the file offset associated with a pipe or FIFO.

[ESRCH] No such process or table entry. No process can be found corresponding to the given process ID.

[ETANOTRUNNING]
Transport agent not running. A transport-agent process for the requested socket is not running in the current processor.

[ETHNOTRUNNING]
OSS terminal helper process is not running. Under normal conditions, the OSS terminal helper process runs in all processors. If this error occurs, follow site-defined procedures for reporting software problems to HP.

[ETIMEDOUT]
Connection timed out. The timer for the connection expired.

[ETXTBSY] Object (text) file busy. The program attempted an operation on a program that is already busy.

[EUNKNOWN]
Unknown error. An unrecognized or very obscure error occurred. If this error occurs, follow site-defined procedures for reporting software problems to HP.

[EVERSION] A version number mismatch exists.

[EWOULDBLOCK]
The operation requested by the program would block.

[EWRONGID] One of the following conditions occurred:

- The process attempted an operation through an operating system input/output process (such as a terminal server process) that has failed or is in the down state.
- The processor for the disk process of the specified file failed during an input or output operation and takeover by the backup process occurred.
- An open file descriptor has migrated to a new processor, but the new processor lacks a resource or system process needed for using the file descriptor.

[EXDEV] Cross-device link. The program attempted to link to a file on another fileset.

[EXDRDECODE]
XDR decoding error. An XDR decoding error occurred.

[EXDRENCODE]
XDR encoding error. An XDR encoding error occurred.

**RELATED INFORMATION**

Functions: **perror(3)**, **strerror(3)**.

**STANDARDS CONFORMANCE**

The following are HP extensions to the XPG4 Version 2 specification:

- The **errno** values [EBADCF], [EBIGDIR], [ECWDTOOLONG], [EDEFINEERR], [EFILEBAD], [EFSBAD], [EFSERR], [EGUARDIANLOCKED], [EGUARDIANOPEN], [EHLDSEM], [EISGUARDIAN], [EMSGQNOTRUNNING], [ENOCPU], [ENOCRE], [ENOIMEM], [ENONSTOP], [ENOROOT], [ENOTOSS], [ENOTSUP], [EOSSNOTRUNNING], [ESPIERR], [ETANOTRUNNING], [ETHNOTRUNNING] [EUNKNOWN], and [EWRONGID] are supported.

## NAME

**filename**, **pathname -** Explains OSS file system file naming

## SYNOPSIS

For OSS files:

*filename*

*pathname*

For local Guardian disk files used from the OSS environment:

**/G/***volume_name*/*subvolume_name*/*file_id*

For local Guardian temporary disk files used from the OSS environment:

**/G/***volume_name*/*temp_file_id*

For local Guardian nondisk devices used from the OSS environment:

**/G/***device_name*/*qualifier*

For remote Guardian disk files used from the OSS environment:

**/E/***node_name***/G/***volume_name*/*subvolume_name*/*file_id*

For remote Guardian temporary disk files used from the OSS environment:

**/E/***node_name***/G/***volume_name*/*temp_file_id*

For remote Guardian nondisk devices used from the OSS environment:

**/E/***node_name***/G/***device_name*/*qualifier*

## PARAMETERS

*filename*
: Identifies a file at a relative location in the OSS file system. A *filename* value can be either a single filename or a symbolic link name.

A single filename is a character string of up to **NAME_MAX** (248) characters, including a null terminator. Valid characters preceeding the null terminator are described under **OSS Filenames** in **DESCRIPTION**, later in this reference page.

A symbolic link name is a string of up to **NAME_MAX** (248) characters, without a null terminator. Valid characters are described under **OSS Filenames** in **DESCRIPTION**, later in this reference page.

A symbolic link name is a pointer to one of the following:

- a single filename
- another symbolic link
- a pathname

*pathname*
: Identifies a file in the OSS file system. A pathname has the following form:

**[ / ]** *filename_1*/*filename_2* **[ / . . .** /*filename_n* **]**

A pathname on a remote NonStop server node begins with **/E/***node_name***/** and is expressed relative to the root directory on that node.

A pathname is a character string of up to **PATH_MAX** (1024) characters, including a null terminator. A pathname consists of one or more filename components, separated by slash (/) characters. Consecutive slashes are interpreted as a single slash.

An absolute pathname begins with a slash character. An absolute pathname identifies an OSS file with respect to the current root directory.

A relative pathname does not begin with a slash character. A relative pathname identifies an OSS file with respect to the current working directory.

The *filename_1* parameter specifies a directory. If *filename_1* is a single period character (**.**, called dot), then *filename_1* indicates the OSS current working directory.

If *filename_1* is two period characters (**..**, called dot-dot), then *filename_1* indicates the parent directory of the OSS current working directory.

The *filename_2* parameter specifies either another directory or the unique identifier for a file other than a directory within the *filename_1* directory. If *filename_2* specifies a directory, then a *filename_3* parameter can be specified, with the same constraints as for *filename_2*, and so on. The last *filename_n* parameter specified must uniquely identify a file that is not a directory.

All *filename_n* parameters must meet the requirements for the *filename* parameter.

*node_name* Specifies the NonStop server node name used by the Expand product for access to files on other nodes. A node name is a character string of up to seven valid characters. Valid characters are the letters a through z and the digits 0 through 9. (Uppercase letters A through Z are accepted but converted to lowercase letters.) The first character must be a letter. Node names specified in the OSS file system do not begin with a backslash (\).

*volume_name* Specifies the disk volume containing the file. A volume name is a character string of up to seven valid characters. Valid characters are the letters a through z and the digits 0 through 9. (Uppercase letters A through Z are accepted but converted to lowercase letters.) The first character must be a letter.

*subvolume_name*
Specifies the disk subvolume (prefix) of the file identifier. A subvolume name is a character string of up to eight valid characters. Valid characters are the letters a through z and the digits 0 through 9. (Uppercase letters A through Z are accepted but converted to lowercase letters.) The first character must be a letter.

*file_id* Specifies the unique identifier of the file within its subvolume. A file identifier is a character string of up to eight valid characters. Valid characters are the letters a through z and the digits 0 through 9. (Uppercase letters A through Z are accepted but converted to lowercase letters.) The first character must be a letter.

*temp_file_id* Specifies the unique identifier of the file within its disk volume. A temporary file identifier is a character string of two to eight valid characters. The first character must be a number sign (**#**). Valid characters for the rest of the string are the digits 0 through 9.

*device_name* Specifies the name of the process providing the interface to the device. This name is a character string of up to seven valid characters. Valid characters are the letters a through z and the digits 0 through 9. (Uppercase letters A through Z are accepted but converted to lowercase letters.) The first character must be a letter.

*qualifier* Specifies a unique identifier significant to the device. A qualifier is a character string of two to eight valid characters. The first character must be a number sign (**#**). Valid characters for the rest of the string are the letters a through z, and the digits 0 through 9. (Uppercase letters A through Z are accepted but converted to lowercase letters.) The second character must be a letter.

## DESCRIPTION

This reference page describes file-naming rules. There is a separate set of naming rules for the file system in each environment:

- Rules for OSS files
- Rules for Guardian files

There are also rules used to translate a name used in one environment to a name valid for the opposite environment.

### OSS Filenames

In the OSS environment, the term "filename" refers to a component of a pathname that contains any characters other than a slash (**/**) character or a null character.

The hyphen (**-**) should not be the first character of an OSS filename if shell commands or utilities will be used on the file. The colon (**:**) should not be a character in an OSS filename if shell commands or utilities will be used on the file.

The OSS file system does not require filename characters to conform to POSIX.1 and ISO C standards for portable filenames. However, the use of portable filenames is strongly recommended.

Valid characters for a portable filename are the letters A through Z, the letters a through z, the digits 0 through 9, and the graphic symbols for period, underscore (**_**), and hyphen (**-**). The hyphen cannot be the first character of a portable filename.

### Guardian Filenames

In the Guardian environment, the term "filename" refers to the set of information that uniquely identifies a Guardian object. A Guardian filename can contain the following characters:

- The letters A through Z (lowercase letters are automatically translated to uppercase and do not appear in the Guardian file system)
- The digits 0 through 9
- The graphic symbols for backslash (\), number sign (**#**), colon (**:**), period, and dollar sign (**$**)

A Guardian filename is approximately equivalent to an OSS pathname; Guardian disk files appear in the **/G** directory with pathnames that have been mapped to OSS filenames.

Guardian objects with Guardian filenames include:

- Disk files
- Temporary disk files
- Nondisk devices
- Named processes

- Unnamed processes

The type of object determines the syntax for the Guardian filename and which subset of the permitted characters is allowed in the parts of that filename.

For a disk file, the Guardian filename consists of the following four parts, separated by periods:

node name
: A character string of two to eight valid characters, specifying the node within the NonStop server Expand network. The first character must be a backslash (\). Valid characters for the rest of the string are the letters A through Z and the digits 0 through 9. The second character must be a letter.

volume name
: A character string of two to eight valid characters, specifying the disk volume containing the file. The first character must be a dollar sign (**$**). Valid characters for the rest of the string are the letters A through Z and the digits 0 through 9. The second character must be a letter.

subvolume name
: A character string of up to eight valid characters, specifying the disk subvolume (prefix) of the file identifier. Valid characters are the letters A through Z and the digits 0 through 9. The first character must be a letter.

file identifier
: A character string of up to eight valid characters, specifying the file. Valid characters are the letters A through Z and the digits 0 through 9. The first character must be a letter.

For a temporary disk file, the Guardian filename consists of the following three parts, separated by periods:

node name
: A character string of two to eight valid characters, specifying the node within the NonStop server Expand network. The first character must be a backslash (\). Valid characters for the rest of the string are the letters A through Z and the digits 0 through 9. The second character must be a letter.

volume name
: A character string of two to eight valid characters, specifying the disk volume containing the file. The first character must be a dollar sign (**$**). Valid characters for the rest of the string are the letters A through Z and the digits 0 through 9. The second character must be a letter.

temporary file identifier
: A character string of two to eight valid characters, specifying the file. The first character must be a number sign (**#**). Valid characters for the rest of the string are the digits 0 through 9.

For a nondisk device, the Guardian filename consists of the following three parts, separated by periods:

node name
: A character string of two to eight valid characters, specifying the node within the NonStop server Expand network. The first character must be a backslash (\). Valid characters for the rest of the string are the letters A through Z and the digits 0 through 9. The second character must be a letter.

device name
: A character string of two to eight valid characters, specifying the name of the process providing the interface to the device. The first character must be a dollar sign (**$**). Valid characters for the rest of the string are the letters A through Z and the digits 0 through 9. The second character must be a letter.

qualifier — A character string of two to eight valid characters, specifying a unique identifier significant to the device. The first character must be a number sign (**#**). Valid characters for the rest of the string are the letters A through Z and the digits 0 through 9. The second character must be a letter.

For a named process, Guardian filename rules are complex. Guardian named processes are not accessible through the OSS file system, so the rules are not discussed here.

Guardian unnamed processes are not accessible through the OSS file system, so identification of them is not discussed here.

Refer to the *Guardian Procedure Calls Reference Manual* for more information about Guardian filenames.

### Translating Guardian Filenames to OSS Filenames/Pathnames

Each portion of a Guardian filename is separately translated to a valid OSS filename. The resulting pathname is prefixed by **/G/** for the local NonStop server node and by **/E/***node_name***/G/** for a remote NonStop server node.

The OSS pathname for the Guardian filename of a disk file therefore becomes **/E/***node_name***/G/***volume_name*/*subvolume_name*/*file_id*. The **/E/** prefix and node name is omitted from the translation for the **/G** directory on the local node.

The following character translations also occur:

- Any dollar sign is deleted.
- Periods are translated to slashes.
- All uppercase letters are translated to lowercase as a normalizing convention. This translation allows filename pattern matching; pattern matching is case-sensitive in the OSS file system.

### Translating OSS Filenames/Pathnames to Guardian Filenames

Each OSS filename within a pathname that includes the **/G** directory is translated to the appropriate part of a fully qualified Guardian filename. The OSS pathname for a file in the **/G** directory cannot contain more OSS filename components than the corresponding Guardian filename permits. Extra components cause an operation to fail; an **errno** value (described under **ERRORS** later in this reference page) is returned for function calls.

For a file on the local NonStop server node, the prefix **/G/** is translated to the local node name. For a file on a remote NonStop server node, the prefix **/E/***node_name***/G/** is translated to the remote node name.

The OSS filenames **.** (dot) and **..** (dot-dot) translate to the corresponding portions of an appropriately resolved OSS pathname when the current working directory is within a **/G** directory.

The following character translations also occur:

- A dollar sign is prefixed to an OSS filename that corresponds to a Guardian volume name or to a Guardian process device name.
- Periods, hyphens (-), and underscores (**_**) within OSS filenames are deleted.
- OSS filenames longer than eight characters are truncated to seven or eight characters, as appropriate under Guardian filename rules.

- Lowercase letters are not translated to uppercase for filename pattern matching. Pattern matching is case-insensitive in the Guardian file system.
- Slashes between OSS filenames are translated to periods.

**EXAMPLES**

1. The following is an example of an absolute OSS pathname:

   **/usr/ccomp/prog1.c**

2. The following is an example of an absolute OSS pathname:

   **/usr/ccomp/prog1.c**

3. The following is an example of an absolute OSS pathname for the file named **prog2** on the remote NonStop server node NODE2:

   **/E/node2/usr/ccomp/prog2.c**

4. The following is a relative OSS pathname for a file in a subdirectory of the current working directory:

   **refman/ch1**

5. The following is an alternative relative OSS pathname for the same file in the same subdirectory of the current working directory:

   **./refman/ch1**

6. The following is a relative OSS pathname for a file in a subdirectory of the parent directory of the current working directory:

   **../yourfiles/oldmail**

7. The following is an example of an absolute OSS pathname for a disk file in the Guardian file system with the Guardian filename \NODE1.$DATA1.MYSUBVOL.MYFILE:

   **/G/data1/mysubvol/myfile**

8. The following is an example of an absolute OSS pathname for a disk file in the Guardian file system on the remote node NODE2 with the Guardian filename \NODE2.$DATA1.MYSUBVOL.MYFILE:

   **/E/node2/G/data1/mysubvol/myfile**

9. The following is an example of an absolute OSS pathname for a temporary disk file in the Guardian file system:

   **/G/guest/#7777777**

10. The following is an example of an absolute OSS pathname for a Guardian nondisk device (a terminal device emulator):

    **/G/ztnt/#pty0001**

11. The following is an example of translating an OSS pathname to a Guardian filename. If the OSS pathname **/G/data.volume/src.l1.v3.4.8/properties.c** is passed to an OSS function call, the OSS pathname is interpreted as if it were specified to be **/G/datavol/srcl1v34/properti**. This interpretation translates to the Guardian filename \LOCAL.$DATAVOL.SRCL1V34.PROPERTI, where \LOCAL is the node name for the local system in the NonStop server Expand network.

12. The following is an example of translating a Guardian filename to an OSS pathname. If \LOCAL.$ZTNT.#PTY0001 is a nondisk device that needs to be passed to an OSS function, it can be referred to by specifying **/G/ztnt/#pty0001**.

**NOTES**

Guardian subvolume names and file identifiers beginning with the letter Z are reserved. Do not use such names for files in the **/G** directory.

During resolution of OSS pathnames, the **st_atime** field of the **stat** structure is updated for each parent directory involved in the resolution.

**ERRORS**

If an invalid Guardian filename results from OSS pathname or OSS filename translation for a file in the **/G** directory, a function call using the OSS pathname or OSS filename fails and one of the following values is returned for **errno**:

[EINVAL] The named file cannot be created.

[ENOENT] The named file cannot be opened.

Most OSS filename and pathname errors return the value [ENAMETOOLONG] for **errno**.

**RELATED INFORMATION**

Commands: **ls(1)**, **mv(1)**, **rm(1)**.

Functions: **creat(2)**, **fstat(2)**, **lstat(2)**, **open(2)**, **stat(2)**, **symlink(2)**.

Files: **limits(4)**.

Miscellaneous: **hier(5)**.

**STANDARDS CONFORMANCE**

The following are HP extensions to the XPG4 Version 2 specification:

- Characters in addition to those of the portable character set are supported in OSS filenames.

## NAME

**hier -** Explains the OSS file system hierarchy

## DESCRIPTION

This reference page describes the file system hierarchy. Subdirectories (and some files) are listed indented after the directory that they appear in.

**/** Root directory of the local OSS file system.

**/bin/** Utility program files, including system and internationalization utilities.

> **unsupported/** Utilities and scripts believed to function correctly but which have not been thoroughly tested and therefore are not supported by HP. Use these utilities and scripts at your own discretion and risk. HP does not guarantee the behavior or performance of these utilities and is not obligated to fix problems associated with them.
>
> **/bin/unsupported/cat1/** contains reference page files for these utilities when reference pages exist for them. Use the command **man -M /bin/unsupported** *utility-name* to read these reference pages.

**/dev/** Device directory, which contains only two devices:

> **tty** Current controlling terminal for the application that is running
>
> **null** Data sink

**/E/** Directories and files for OSS and Guardian file systems on remote NonStop server nodes accessible through the Expand product. Do not mount filesets or create files here.

**/etc/** System configuration files (such as the default **profile**, **termcap**, and **printcap** files) and sockets-related network configuration files (such as the site-modified **hosts**, **networks**, **protocols**, and **services** files). These are not executable files.

> **install_obsolete/**
> Files containing lists of files from earlier releases that HP recommends you remove from your system. These files can be used as input for the **Pcleanup** utility.

**/G/** Guardian files. For example, the Guardian file $SYSTEM.SYSTEM.SCF ($*volume.subvolume.fileid*) is stored as **/G/system/system/scf**.

Each Guardian volume is a separate fileset. Guardian environment processes also appear in this directory. See the **stat(2)** reference page for additional information.

**/lost+found/** Files located by the fileset checking program of the OSS Monitor. These have names of the form #*inode_number*, where *inode_number* identifies the inode number that is associated with the recovered file within the OSS file system.

**/nonnative/** Files for use with G-series TNS or accelerated applications.

> **bin/** G-series TNS or accelerated files corresponding to the native files found in **/bin/**. The accelerated version of the **c89** utility and the TNS C compiler are located here on G-series systems.

**usr/** G-series TNS or accelerated files corresponding to the native files found in **/usr/**.

**share/man/** Reference page files for use with the **apropos**, **man**, and **whatis** commands when the corresponding G-series TNS or accelerated function or utility cannot be described in the reference page files found in **/usr/share/man/**.

Use the command **man -M /nonnative/usr/share/man** *topic* to read these reference pages.

**/tmp/** System-generated temporary files (the contents of **/tmp** are usually not preserved across a system reboot).

**/usr/** User utilities and applications:

**bin/** Native language and tools utilities, including the native **c89** utility

**include/** C program header (include) files

**lib/** C run-time library routines, internationalization message catalogs, locale conversion files, compiler components, and shared resource libraries

**share/** Text files, such as reference (man) page files

**man/** Initially, the **man/** directory contains:

**cat1/** through **cat8/**
Preformatted Open System Services reference pages

**whatis.frag/** The separately maintained portions of the **whatis** database

**/var/** Files that increase in size until the size is deliberately reduced. Normally contains log files and similar files to which information is periodically appended.

**RELATED INFORMATION**

Commands: **find(1)**, **grep(1)**, **ls(1)**.

Functions: **stat(2)**.

Files: **null(7)**.

## NAME

**login.defs -** The default login configuration file for the user management suite on OSS.

## DESCRIPTION

The **/etc/login.defs** file contains the default values for login configurable variables used in the user and alias management utilities. The default values are used in the absence of user-specified values. The **login.defs.sample** file provides examples for setting variables.

## FILE FORMAT

The file consists of variable, value duple entries. They are of the form <variable name> <value>, specified one duple entry per line. Blank lines and comment lines are ignored. Comments begin with the "#" sign as the first non-white character of the line.

## SECTION DESCRIPTIONS

The utilities in the user management suite recognize following variables:

CREATE_HOME, PASS_MAX_DAYS, USERDEL_CMD, and UMASK.

## RELATED INFORMATION

**users (5), newusers(8), useradd(8), userdel(8), usermod(8).**

## NAME

**pathname -** Explains OSS file system path naming

## DESCRIPTION

See the **filename(5)** reference page.

## NAME

**process_extension_results -** Contains the status of a process creation attempt

## SYNOPSIS

**#include <tdmext.h>**

**struct process_extension_results** ***pr_results*;**

## PARAMETERS

*pr_results* Points to the output structure containing optional process identification and error information. In case of error, this structure provides additional information including the PROCESS_LAUNCH_ procedure error and error detail. The structure is defined in the **tdmext.h** header file.

The structure must be defined locally and initialized before its first use. Initialization is done using the **#define DEFAULT_PROCESS_EXTENSION_RESULTS**, as defined in the **tdmext.h** header file.

## DESCRIPTION

The **process_extension_results** structure contains status information after a call to the **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, or **tdm_spawnp( )** function. This output structure is also used by Guardian environment procedures such as PROCESS_SPAWN_; therefore, some returned values defined in the **tdmext.h** header file are not returned to an OSS process and are not described in this reference page.

Not all of the returned values described in the following subsection are returned for a call to a specific OSS function.

The **tdmext.h** header file is not kept current when new error codes are defined for process creation functions. The list of **_TPC_** macros described in this reference page is not complete; for a current description of error macros and error codes, see the Guardian header file $SYSTEM.ZSPIDEF.ZGRDC or the summary of process-creation errors in the *Guardian Procedure Calls Reference Manual* (see the table entitled "Summary of Process Creation Errors").

### Output Structure Information

If the *pr_results* parameter does not contain a null pointer, it points to an output structure defined in the **tdmext.h** header file. This structure can contain fields that vary from release to release, including reserved and filler fields.

First, the output structure must be initialized using the **#define DEFAULT_PROCESS_EXTENSION_RESULTS**. This initialization sets the value of the **pr_len** field to the correct value for the current release. The value of the **pr_len** field should not be modified after being set by **#define DEFAULT_PROCESS_EXTENSION_RESULTS**.

The following fields are meaningful in the current release:

```
#ifdef  __LP64

    typedef struct process_extension_results {
        int         pr_len;
        short       pr_phandle[10];
        int         pr_pid;
        int         pr_errno;
        short       pr_TPCerror;
        short       pr_TPCdetail;
     }  process_extension_results_def;

#else  /*  !  __LP64  */
```

```
typedef struct process_extension_results {
    long        pr_len;
    short       pr_phandle[10];
    long        pr_pid;
    long        pr_errno;
    short       pr_TPCerror;
    short       pr_TPCdetail;
} process_extension_results_def;
#endif /* ! __LP64 */
```

## RETURN VALUES

Upon successful completion of the function call, this structure returns the following information:

**pr_len** Specifies the size in bytes of the structure. This value is the one specified as input.

**pr_phandle** Contains the Guardian process handle of the new process.

**pr_pid** Contains the OSS process ID of the new process.

**pr_errno** Contains the OSS error number normally returned in **errno**.

**pr_TPCerror** Identifies the process creation error. If the *pr_results* parameter of the function call did not contain a null pointer, the structure it points to returns additional error information including the PROCESS_LAUNCH_ error and error detail.

Refer to the **ERRORS** section of this reference page for a description of the returned information.

**pr_TPCdetail** Contains additional error information, as indicated in the descriptions of **pr_TPCerror** field values. Refer to the **ERRORS** section of this reference page for a description of the returned information.

## ERRORS

When an error occurs and the calling function provided a nonnull pointer for its *pr_results* parameter, one of the following values is returned in **pr_TPCerror**:

0 No error information is available. The contents of the **pr_TPCdetail** field are not meaningful.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_PARAM_REFERENCE**

A pointer value in a field in the structure pointed to by the *pe_parms* parameter of the calling function is invalid. Refer to error 3 in the discussion of PROCESS_LAUNCH_ errors in the *Guardian Procedure Errors and Messages Manual* for more information.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

The **pr_TPCdetail** field contains one of the following values to provide additional information:

**_TPC_BAD_ARGV**

The pointer to the *argv***[ ]** array parameter of the calling function or one of the entries in the array is invalid.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_spawn( )**,

**tdm_spawnp( )**.

**_TPC_BAD_DEFINES**

The pointer to the **pe_defines** field in the structure pointed to by the *pe_parms* parameter of the calling function is invalid.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_ENVIRON**

One of the pointers in the **environ** array is invalid.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_ENVP**

The pointer to the *envp***[ ]** array parameter of the calling function or one of the entries in the array is invalid.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_ERROR_DETAIL**

An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**.

**_TPC_BAD_EXTENSION**

The structure pointed to by the *pe_parms* parameter of the calling function and used in the function call is invalid.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_EXTSWAP**

The pointer to the **pe_extswap_file_name** field in the structure pointed to by the *pe_parms* parameter of the calling function is invalid.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_FDMAP**

The *fd_map***[ ]** array parameter of the calling function is not valid for *fd_count* elements.

Issued for: **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_HOMETERM**

The pointer to the **pe_hometerm** field in the structure pointed to by the *pe_parms* parameter of the calling function is invalid.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_INHERIT**

The pointer to the *inherit* structure parameter of the calling function is invalid.

Issued for: **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_INTERNAL**

An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_OUTPUT**

An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**.

**_TPC_BAD_OUTPUT_LEN**

An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**.

**_TPC_BAD_PARMLIST**

An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_PIMFILE**

An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_PRIVATE_LIST**

An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_PRIVLIST**

An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_PROCESS_NAME**

The pointer to the **pe_process_name** field in the structure pointed to by the *pe_parms* parameter of the calling function is invalid.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_SWAP**

The pointer to the **pe_swap_file_name** field in the structure pointed to by the *pe_parms* parameter of the calling function is invalid.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_UC**

The *file* parameter pointer of the calling function is invalid.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_UL**

The pointer to the **pe_library_name** field in the structure pointed to by the *pe_parms* parameter of the calling function is invalid.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_PARAM_VALUE**

Some combination of values for fields in the structure pointed to by the *pe_parms* parameter of the calling function is invalid. Refer to error 2 in the discussion of PROCESS_LAUNCH_ errors in the *Guardian Procedure Errors and Messages Manual* for more information.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

The **pr_TPCdetail** field contains one of the following values to provide additional information:

**_TPC_BAD_CPU**

The **pe_cpu** field in the structure pointed to by the *pe_parms* parameter of the calling function contains an invalid value.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_CREATE_OPTIONS**

The **pe_create_options** field in the structure pointed to by the *pe_parms* parameter of the calling function contains an invalid value.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_DEBUG_OPTIONS**

The **pe_debug_options** field in the structure pointed to by the *pe_parms* parameter of the calling function contains an invalid value.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_DEFINES**

The **pe_defines** field in the structure pointed to by the *pe_parms* parameter of the calling function is invalid.

Issued for: **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_EXTENSION**

The structure pointed to by the *pe_parms* parameter of the calling function and used in the function call is invalid.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_EXTSWAP**

The **pe_extswap_file_name** field in the structure pointed to by the *pe_parms* parameter of the calling function contains an invalid OSS pathname for a Guardian file.

This error occurs only for G-series TNS or accelerated new process image files.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_HOMETERM**

The **pe_hometerm** field in the structure pointed to by the *pe_parms* parameter of the calling function points to an invalid Guardian process name. Refer to error 14 in the discussion of file-system errors in the *Guardian Procedure Errors and Messages Manual* for more information.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_INHERIT**

One of the fields in the *inherit* structure parameter of the calling function is invalid.

Issued for: **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_INTERNAL**

An internal OSS software problem occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_INTERPRETER**
The shell script contained in the text file pointed to by the *file* parameter of the calling function does not have an interpreter name in its **#!** header line.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_JOB**
The **pe_jobid** field in the structure pointed to by the *pe_parms* parameter of the calling function contains an invalid value.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_MEM**
The **pe_memory_pages** field in the structure pointed to by the *pe_parms* parameter of the calling function contains an invalid value.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_NAME_OPTIONS**
The **pe_name_options** field in the structure pointed to by the *pe_parms* parameter of the calling function contains an invalid value.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_OSS_OPTIONS**
The **pe_OSS_options** field in the structure pointed to by the *pe_parms* parameter of the calling function contains an invalid value.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_OUTPUT**
An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_spawn( )**.

**_TPC_BAD_OUTPUT_LEN**
An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_spawn( )**.

**_TPC_BAD_PARMLIST**
An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_PFS_SIZE**

The **pe_pfs_size** field in the structure pointed to by the *pe_parms* parameter of the calling function contains an invalid value.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_PIMFILE**

An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_PRIO**

The **pe_priority** field in the structure pointed to by the *pe_parms* parameter of the calling function contains an invalid value.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_PRIVATE_LIST**

An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_PRIVLIST**

An internal OSS software error occurred. If the problem persists, follow site-defined procedures for reporting software problems to HP.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_PROCESS_NAME**

The **pe_process_name** field in the structure pointed to by the *pe_parms* parameter of the calling function points to an invalid Guardian process name.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_SWAP**

The **pe_swap_file_name** field in the structure pointed to by the *pe_parms* parameter of the calling function points to an invalid Guardian swap file name.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_UC**

The *file* parameter value of the calling function cannot be resolved into a valid program file name.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

**_TPC_BAD_UL**

The **pe_library_name** field value of the calling function cannot be resolved into a valid library file name.

Issued for: **tdm_execve( )**, **tdm_execvep( )**, **tdm_fork( )**, **tdm_spawn( )**, **tdm_spawnp( )**.

For information about specific Guardian file-system errors, see the discussion of file-system errors in the *Guardian Procedure Errors and Messages Manual*.

## RELATED INFORMATION

Functions: **tdm_execve(2)**, **tdm_execvep(2)**, **tdm_fork(2)**, **tdm_spawn(2)**, **tdm_spawnp(2)**.

## STANDARDS CONFORMANCE

This structure is an extension to the XPG4 Version 2 specification.

## NAME

**resolv.conf -** BIND 9 Domain Name System resolver configuration file

## DESCRIPTION

The default configuration file **/etc/resolv.conf** provides an explicit default domain name for the Domain Name System (DNS) to use, and identifies name servers on other processors. The BIND 9 resolver system can be used with other, nondefault versions of **resolv.conf**.

Each entry in a **resolv.conf** file is a directive that consists of a keyword followed by one or more values:

*keyword* *value*

The **keyword** and *value* must appear on a single line. Start the line with the keyword, followed by the value, separated by white space.

The **/etc/resolv.conf** file can contain the following directives:

**nameserver** *address*
: The Internet address of a domain name server, in standard dot notation. Up to **MAXNS** multiple domain name server addresses may be listed.

  The resolver queries the domain name servers in the order they are listed in the file, stopping when it receives a response, or moving to the next in the list if the query times out. If the resolver reaches the end of the domain name server list without receiving a response, it will start from the beginning of the list and query each domain name server again, until a maximum number of retries is reached. If **/etc/resolv.conf** contains no **nameserver** directives, the resolver uses the loopback address. Therefore, a domain name server must be running on the processor on which the file resides.

**domain** *name*
: The default domain to append to names that do not contain a domain, and the default domain name to be used in searches. No trailing spaces are allowed after the value in *name*.

  If **resolv.conf** does not contain a **domain** directive, then the resolver uses the the hostname for the processor, but removes the first part of the name. For example, if the host name is set to "`yojimbo.dev1.anyfirm.com`," the resolver uses the name "`dev1.anyfirm.com`."

  Most queries for names within this domain can use short names relative to the local domain. (Everything after the first "**.**" is presumed to be the domain name.) If the hostname does not contain a domain part, the root domain is assumed. You can use the **LOCALDOMAIN** environment variable to override the domain name.

  The **domain** and **search** keywords are mutually exclusive. If more than one instance of these keywords is present, the last instance takes precedence.

**search** *name* ...
: The explicit search order that you want the resolver to use when looking up hostnames. The **search** keyword can accept up to six domain names as values, with a total of 256 characters.

  The resolver will perform its search using the order specified after the **search** keyword.

  The search list is normally determined from the local domain name. By default, it contains only the local domain name. You can change the default behavior by listing the desired domain search path following the **search** keyword, with spaces or tabs separating the names. Most resolver queries are attempted using each component of the search path in turn until a match is found. This process might be slow and generates a lot of network traffic if the servers for the listed domains are not local. Queries time out if no server is available for one of the domains.

The **domain** and **search** keywords are mutually exclusive. If more than one instance of these keywords is present, the last instance takes precedence.

**sortlist** *addresslist*

Allows addresses returned by internal function calls to be sorted.

An *addresslist* is specified by IP address netmask pairs. The netmask pairs are optional; an unspecified netmask defaults to the natural netmask of the net. The IP address and optional netmask pairs are separated by slashes. Up to 10 netmask pairs may be specified.

Example: **sortlist 130.155.160.0/255.255.240.0 130.155.0.0**

**options** *option*...

Allows internal resolver variables to be modified. Possible values for *option* are:

**debug** Sets **RES_DEBUG** in the **_res.options** field.

**ndots:***n* Sets a threshold floor for the number of dots which must appear in a name before an initial absolute (as-is) query is performed. The default value for *n* is 1, which means that if there are any dots in a name, the name is tried first as an absolute name before any search list elements are appended to it.

**timeout:***n* Sets the amount of time the resolver waits for a response from a remote name server before retrying the query via a different name server. Measured in seconds, the default is **RES_TIMEOUT** (described in the **<resolv.h>** header file).

**attempts:***n* Sets the number of times the resolver sends a query to its name servers before giving up and returning an error to the calling application. The default is **RES_DFLRETRY** (described in the **<resolv.h>** header file).

**no-check-names**

Sets **RES_NOCHECKNAME** in **_res.options**. This disables the modern BIND checking of incoming host names and mail names for invalid characters such as underscore (**_**), nonASCII, or control characters.

You can override the **search** keyword of the **resolv.conf** file on a per-process basis by setting the environment variable **LOCALDOMAIN** to a space-separated list of search domains.

You can amend the **options** keyword of the **resolv.conf** file on a per-process basis by setting the environment variable **RES_OPTIONS** to a space-separated list of resolver options.

## EXAMPLES

Example lines from a **/etc/resolv.conf** file are shown below:

```
domain          dev1.anyfirm.com
nameserver      123.456.78.90
nameserver      123.456.78.91
```

## RELATED INFORMATION

Commands: **dnssec_named(8)**, **named(8)**.

Functions: **gethostbyaddr(3)**, **gethostbyname(3)**, **gethostbyname2(3)**, **setnetent(3)**.

Files: **hosts(4)**, **networks(4)**, **protocols(4)**, **resolv.conf(4)**, **services(4)**.

**NAME**

users - The default user configuration file for the user management suite on OSS.

**DESCRIPTION**

The **/etc/default/users** file contains the default values for configurable variables used in the user and alias management utilities. The default values are used in the absence of user-specified values. You must create and customize this file for your site-specific needs. The **users.sample** file provides examples for setting variables.

**FILE FORMAT**

The file consists of variable, value duple entries. They are of the form <variable name>=<value>, specified one duple entry per line. Blank lines and comment lines are ignored. Comments begin with the "#" sign as the first non-white character of the line.

**SECTION DESCRIPTIONS**

The utilities in the user management suite recognize following variables:

GROUP, USER, HOME, EXPIRE, INACTIVE, and SKEL.

**RELATED INFORMATION**

**login.defs(5), newusers(8), useradd(8), userdel(8), usermod(8).**

# Permuted Index

# Index

## Symbols



## A



## B



## C

# D



# E

# F

# G

# H



# I



# K



# L

# M

# N



# O



# P

# R

# S

# T



# U



# V

# W