HP NonStop TCP/IP Programming Manual

HP Part Number: 524521-020
Published: March 2014
Edition: J06.04 and subsequent J-sries RVUs, H06.03 and subsequent H-series RVUs, G06.00 and subsequent G-series RVUs, D40.00 and subsequent D-series RVUs

Contents

About This Document

This manual describes application development for the NonStop TCP/IP, Parallel Library TCP/IP, NonStop TCP/IPv6, and CIP subsystems using the HP Guardian socket library routines.

Supported Release Version Updates (RVUs)

TCP/IP: D40.00 and all subsequent D-series RVUs, G06.00 and all subsequent G-Series RVUs, and H06.03 and all subsequent H-series RVUs until otherwise indicated by its replacement publication

Parallel Library TCP/IP: G06.08 and all subsequent G-series RVUs until otherwise indicated by its replacement publication

NonStop TCP/IPv6: G06.20 and all subsequent G-series RVUs, H06.05 and all subsequent H-series RVUs until otherwise indicated by its replacement publication

Cluster I/O Protocols (CIP): J06.04 and all subsequent J-series RVUs until otherwise indicated by its replacement publication

Intended Audience

This manual is intended for experienced C and TAL programmers. You must be familiar with the following protocols and products:

- The standard TCP/IP family of protocols described in various Requests for Comments (RFCs)
- The Berkeley socket interface
- Use of NonStop systems, including the HP NonStop operating system

New and changed information for March 2014 (524521-020)

This edition of the manual includes the following changes:

- Changed “address” word to “value” for flags “AI_NUMERICHOST” (page 64) and “AI_NUMERICSERV” (page 64).
- Added “Note” in the section “socket_set_inet_name” (page 200).

New and changed information for February 2013 (524521-019)

This edition of the manual includes the following changes:

- Added a new note in the “Usage Guidelines” (page 103) section.

New and changed information for July 2012 (524521-018)

This edition of the manual includes the following changes:

- Added the function details and usage consideration in the “accept_nw3” (page 97) section.
- Added the usage guidelines for the functions `gethostbyname` and `host_file_gethostname`“Usage Guidelines” (page 111).
- Added new guideline for the section “Usage Guidelines” (page 108).

New and changed information for February 2012 (524521-017)

This edition of the manual includes the changes to enable 64–bit support:

- Added the 64–bit APIs, send64_ (page 168), sendto64_ (page 179), send_nw64_ (page 171), send_nw2_64_ (page 175), recv64_, recv_nw64_ (page 155), recvfrom_nw64_ (page 164),

recvfrom64_ (page 160), t_sendto_nw64_, t_recvfrom_nw64_ (page 203) and sendto_nw64_ (page 182).

- Changed the data type of length parameters to `socklen_t` in `inet_ntop`, `getnameinfo`, `gethostname`, `gethostname`, `lwres_getipnodebyaddr` and `lwres_getnameinfo` APIs.

New and changed information for August 2011 (524521-016)

This edition of the manual includes the following changes:

- Added hostname and IP address resolution in the "Domain Name Resolution" (page 26) section.
- Updated text for SO_REUSEPORT in the "setsockopt, setsockopt_nw" (page 184) section.
- Added TCP^RESOLVER^ORDER description in "Using the DEFINE Command" (page 29) table.

New and Changed Information for October 2010 (524521-015)

This edition of the manual includes changes to the usage guidelines in:

- "accept_nw" (page 91)
- "accept_nw1" (page 94)
- "setsockopt, setsockopt_nw" (page 184)

Changes and Additions for September 2010 (524521-014)

This edition of the manual included changes to the usage guidelines in:

- "accept_nw" (page 91)
- "accept_nw1" (page 94)
- "accept_nw2" (page 95)
- "setsockopt, setsockopt_nw" (page 184)

Changes and Additions for March 2010 Update (524521-013)

Changes in the -013 edition of the manual include:

- A missing error definition was added to the send (page 166) routine.
- Information regarding rogue clients was added to "Usage Guidelines" for the "bind, bind_nw" (page 98) routines.
- The input value was updated for `SO_ERROR` in the "setsockopt, setsockopt_nw" (page 184) routines.
- Corrected the protocol listed for the ntp service in "Port Numbers for Host-Specific Functions" (page 242).
- Updated "Client and Server Programs Using UDP" (page 219) to describe how to use NonStop TCP/IPv6 to call the socket_ioctl function, including configuring the Provider attribute for an address family.
- Updated several socket error definitions for "Socket Errors" (page 243).

Changes and Additions for September 2008 Update (524521-012)

This edition of the manual has been updated to reflect support for Cluster I/O Protocols (CIP).

Other changes include:

- Descriptions of the "t_recvfrom_nw" (page 201) and "t_sendto_nw" (page 204) socket routines, removed in an earlier edition, have been restored.
- The description of "sock_close_reuse_nw" (page 190) has been updated to describe error 4123.
- IPV6_V6ONLY has been added to the descriptions of the "getsockopt, getsockopt_nw" (page 128) and "setsockopt, setsockopt_nw" (page 184) routines.
- Library routine parameters have been identified as input or return values in their definitions.

Changes and Additions January 2007 Update (524521-010)

Changes in this edition of the manual include:

- Missing error definitions were added to accept_nw (page 91).
- A missing error definition was added to send_nw2 (page 173).
- Corrections were made to recvfrom (page 158) and recvfrom_nw (page 161).

Changes and Additions for the H06.05 RVU (February 2006, 524521-009)

Updates in this edition show that the lightweight resolver library calls are supported on H06.05 and later H-series RVUs. (See Chapter 4 (page 81).)

Hyperlinks in New and changed information for February 2012 (524521-017) for previous editions fixed. (See "Changes and Additions for the H06.03 RVU (July, 2005 524521-006)" (page 13).)

A note has been added to the new and changed library routines in Chapter 4 (page 81) to indicate that they are only supported on G06.27 and later G-series RVUs and are not supported on H06.03 and later H-series RVUs until otherwise indicated in a replacement edition.

Changes and Additions for the G06.27 RVU (September 2005, 524521-007)

Twelve new functions have been added to NonStop TCP/IPv6 to support the lightweight resolver library for DNS. These new functions are:

- gethostbyname2 (page 112)
- lwres_freeaddrinfo (page 140)
- lwres_freehostent (page 141)
- lwres_gai_strerror (page 141)
- lwres_getaddrinfo (page 142)
- lwres_gethostbyaddr (page 144)
- lwres_gethostbyname (page 145)
- lwres_gethostbyname2 (page 146)
- lwres_getipnodebyaddr (page 147)
- lwres_getipnodebyname (page 149)
- lwres_getnameinfo (page 150)
- lwres_hstrerror (page 152)

Changes and Additions for the H06.03 RVU (July, 2005 524521-006)

Information about the HP Integrity NonStop NS-series server was added to this edition of the manual.

In addition, the following corrections were made:

- Sample programs were modified to show Include statements.
- TAL syntax diagrams were updated to show INT(32) declarations instead of INT since the WIDE model is more frequently used.
- The description of `inet_ntop` inTable 13 (page 83) was corrected.
- There was a correction to the code example for freeaddrinfo (page 104).
- Since you no longer have to define the SRL before starting the TCP6SAM process (as of G06.24), the table of DEFINEs in the section Using the DEFINE Command (page 29) was changed to reflect that the SRL only needs to be defined for TCPSAM processes and TCP6SAM processes for pre-G06.24 RVUs of NonStop TCP/IPv6.
- Statements for including the appropriate header files were added to the syntax declarations for the data structures shown in Chapter 3 (page 62).
- TAL definitions for library-routine syntax in Chapter 4 (page 81) were modified wherever INT declarations were made so that INT(32) is shown instead.
- A correction was to the errors defined for if_nameindex (page 132).
- The introductory paragraph for the example for if_freenameindex (page 130) was corrected to refer to the `if_nameindex` function demonstrated in the sample.

Correction Update (December 2004, 524521-005)

- The TAL synopsis for the sock_close_reuse_nw library routine was added under sock_close_reuse_nw (page 190).
- The description of the flags parameter of the socket, socket_nw (page 191) was modified.
- The usage guidelines of the socket, socket_nw library routine was modified under Usage Guidelines (page 193)Usage Guidelines.
- A sample program for AF_INET No-Wait Server Stub Routine was added underAF_INET No-Wait Server Stub Routine (page 212).
- These sample programs have been modified to run without warnings:
  - AF_INET Server Stub Routine (page 210)
  - C TCP Client Program (page 215)
  - C TCP Server Program (page 217)
  - UDP Client Program (page 219)
  - UDP Server Program (page 223)

Correction Update (September 2004, 524521-004)

Information has been added to the error descriptions for accept_nw2 (page 95).

Manual Consolidation Update (March 2004, 524521-003)

- Information about the Parallel Library TCP/IP subsystem has been added to this manual; all three NonStop TCP/IP subsystems are now documented in this manual and the *TCP/IP and IPX/SPX Programming Manual* has been changed to the *IPX/SPX Programming Manual*.
- Overview information about the three NonStop TCP/IP subsystems has been added to NonStop TCP/IP Subsystems and the Guardian Sockets Application Program Interface (API) (page 23).

- Sample TCP/IP programs have been moved to this manual from the *TCP/IP and IPX/SPX Programming Manual* in Chapter 5 (page 208).
- Other minor changes have been made to the manual to incorporate the Parallel Library TCP/IP subsystem.

G06.22 RVU Update (December 2003, 524521-002)

- Information about using sockets in both the conventional NonStop TCP/IP and NonStop TCP/IPv6 environments has been added. (See Using NonStop TCP/IP and NonStop TCP/IPv6 or Parallel Library TCP/IP (page 24).)
- The limitations of raw-socket support for NonStop TCP/IP have been documented. (See Programmatic Interface to Raw Sockets (page 41).)
- Information about using the new logical network partitioning feature has been added. (See Multiple NonStop TCP/IP Processes and Logical Network Partitioning (LNP) (NonStop TCP/IPv6, H-Series and G06.22 and Later G-Series RVUs Only) (page 43) and accept_nw2 (page 95).)
- Procedures for determining process names has changed. (See Process Names (page 43).)
- New TCP retransmission timers have been documented (getsockopt, getsockopt_nw (page 128) and setsockopt, setsockopt_nw (page 184)).
- The buffer size for `SO_RCVBUF` and `SO_SNDBUF` has been corrected for NonStop TCP/IPv6. (See Usage Guidelines for setsockopt, setsockopt_nw (page 184).)
- Considerations for the use of `sock_close_reuse_nowait` have been added (sock_close_reuse_nw (page 190)).
- The `setsockopt` level definitions have been reorganized to separate `IPROTO_IP` and `IPPROTO_IPV6` (setsockopt, setsockopt_nw (page 184)).
- More information has been added to the error message EACCES (4013) in Appendix B (page 243). In addition, this error has been added to sendto (page 177) and sendto_nw (page 180).
- Use of the word subnet has been clarified to distinguish between the generic-networking term and the NonStop TCP/IPv6 SCF object. See Notation for Subnet (page 19).

G06.20 RVU Update (May 2003, 524521-001)

This manual was new for the G06.20 RVU.

Document Organization

This document is organized as follows:

- Chapter 1 (page 23) provides an overview of the three HP NonStop TCP/IP subsystems, some TCP/IP fundamentals, considerations for programming in the Guardian environment, and information about multicasting and multiplexing.
- Chapter 2 (page 49) provides procedures for porting your applications for IPv6 use or protocol-independence and procedures for developing new IPv6 applications.
- Chapter 3 (page 62) provides the definitions of the Guardian sockets library data structures.
- Chapter 4 (page 81) provides the definitions and usage guidelines for the Guardian sockets library routines.
- Chapter 5 (page 208) provides sample server and client code for both IPv4 and IPv6.
- Appendix A (page 241) lists the protocol numbers most commonly used with the raw socket (IP) interface, together with the names that you can use for these protocols in programs.
- Appendix B (page 243) describes the error conditions for the socket routines and explains how a program can recover from the errors.

Notation Conventions

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS

Uppercase letters indicate keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

MAXATTACH

Italic Letters

Italic letters, regardless of font, indicate variable items that you supply. Items not enclosed in brackets are required. For example:

file-name

Computer Type

Computer type letters indicate:

- C and Open System Services (OSS) keywords, commands, and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

  Use the `cextdecs.h` header file.

- Text displayed by the computer. For example:

```
Last Logon: 14 May 2006, 08:02:23
```

- A listing of computer code. For example

```
if (listen(sock, 1) < 0)
{
perror("Listen Error");
exit(-1);
}
```

Bold Text

Bold text in an example indicates user input typed at the terminal. For example:

```
ENTER RUN CODE

?123
CODE RECEIVED:      123.00
```

The user must press the Return key after typing the input.

[ ] Brackets

Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name

INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
FC [ num  ]
   [ -num ]
   [ text ]

K [ X | D ] address
```

{ } Braces

A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }

ALLOWSU { ON | OFF }
```

| Vertical Line

A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis

An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]…

 - ] {0|1|2|3|4|5|6|7|8|9}…
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char…"
```

Punctuation

Parentheses, commas, semicolons, and other symbols not previously described must be typed as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;

LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must type as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing

Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In this example, no spaces are permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing

If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE

   [ , attribute-spec ]…
```

!i and !o

In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                    !i
                         , error        ) ;              !o
```

!i,o

In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                      !i,o
```

!i:i

In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length           !i:i
                            , filename2:length ) ;       !i:i
```

!o:i

In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                        !i
                        , [ filename:maxlen ] ) ;        !o:i
```

Notation for Messages

This list summarizes the notation conventions for the presentation of displayed messages in this manual.

Bold Text

Bold text in an example indicates user input typed at the terminal. For example:

```
ENTER RUN CODE

?123
CODE RECEIVED:      123.00
```

The user must press the Return key after typing the input.

Nonitalic Text

Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

Italic Text

Italic text indicates variable items whose values are displayed or returned. For example:

```
p-register

process-name
```

[ ] Brackets

Brackets enclose items that are sometimes, but not always, displayed. For example:

```
Event number = number [ Subject = first-subject-value ]
```

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
proc-name trapped [ in SQL | in SQL file system ]
```

{ } Braces

A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
obj-type obj-name state changed to state, caused by
{ Object | Operator | Service }

process-name State changed from old-objstate to objstate
{ Operator Request. }
{ Unknown.          }
```

| Vertical Line

A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
Transfer status: { OK | Failed }
```

% Percent Sign

A percent sign precedes a number that is not in decimal notation. The % notation precedes an octal number. The %B notation precedes a binary number. The %H notation precedes a hexadecimal number. For example:

```
%005400

%B101111

%H2F

P=%p-register E=%e-register
```

Notation for Subnet

The following describes the notation conventions for SUBNET and subnet used in this manual.

UPPERCASE LETTERS

Uppercase letters indicate the NonStop TCP/IP, Parallel Library TCP/IP or NonStop TCP/IPv6 SCF SUBNET object. For example:

Port A is identified by logical interface (LIF) 018, which uses a SUBNET on the TCP/IP process named $ZB018 in processor 0.

lowercase letters

Lowercase letters indicate the general networking term for subnet. For example:

Multicast datagrams that have a Time-To-Live (TTL) value of 1 are forwarded only to hosts on the local subnet.

Notation for Management Programming Interfaces

This list summarizes the notation conventions used in the boxed descriptions of programmatic commands, event messages, and error lists in this manual.

UPPERCASE LETTERS

Uppercase letters indicate names from definition files. Type these names exactly as shown. For example:

```
ZCOM-TKN-SUBJ-SERV
```

lowercase letters

Words in lowercase letters are words that are part of the notation, including Data Definition Language (DDL) keywords. For example:

```
token-type
```

!r

The !r notation following a token or field name indicates that the token or field is required. For example:

```
ZCOM-TKN-OBJNAME      token-type ZSPI-TYP-STRING.          !r
```

!o

The !o notation following a token or field name indicates that the token or field is optional. For example:

```
ZSPI-TKN-MANAGER      token-type ZSPI-TYP-FNAME32.         !o
```

Related Information

If you are writing programs that use the socket routines described in this manual, you should refer to the following manuals:

- *TCP/IPv6 Configuration and Management Manual* for complete descriptions of NonStop TCP/IPv6, including file formats and other specific information that applies to the whole subsystem. This manual also describes the Subsystem Control Facility (SCF) interactive interface that allows operators and system managers to configure, control, and monitor the NonStop TCP/IPv6/IP subsystem.
- *TCP/IP Configuration and Management Manual* for information about the architecture and management of the NonStop TCP/IP subsystem.
- *TCP/IP (Parallel Library) Configuration and Management Manual* for information about the architecture and management of the Parallel Library TCP/IP subsystem.
- *LAN Configuration and Management Manual* for descriptions of the SLSA subsystem, which provides parallel LAN I/O for NonStop S-series systems. In particular, this manual provides information about logical interfaces (LIFs) and physical interfaces (PIFs) which are key concepts for NonStop TCP/IP, Parallel Library TCP/IP, and NonStop TCP/IPv6.
- *TCP/IP Applications and Utilities User Guide* describes the interactive interfaces to the following TCP/IP applications: ECHO, FINGER, FTP, LISTNER, TFTP, TELNET, and TN6530. Server information is included for FTP, TFTP, and TELNET.

If you are writing programs that use the socket function calls described in this manual, read the following manuals for background and reference information:

- The *C/C++ Programmer's Guide* provides information about the HP C language and compiler, including the supplementary functions for the NonStop operating system environment.
- The *TAL Reference Manual* provides information about the HP TAL language and compiler.
- The *TAL Programmer's Guide* provides information on mixed-language programming.
- The *CRE Programming Manual* provides information about programming sockets in the Common Run-Time Environment (CRE) using the HP TAL language and compiler.

- The *Guardian Programmer's Guide* describes how to program in the NonStop operating system environment.
- The *Guardian Procedure Calls Reference Manual* lists the syntax and semantics of the NonStop system procedure calls whose functions are not available in the HP C language.
- The *Guardian Procedure Errors and Messages Manual* describes the Guardian messages for NonStop systems that use the NonStop operating system.
- The *HP NonStop Kernel Programmer's Guide* provides information on programming for the NonStop operating-system environment.
- The *TCP/IPv6 Migration Guide* provides a comparison of NonStop TCP/IPv6, NonStop TCP/IP and Parallel Library TCP/IP.

Publishing History

Part Number	Product Version	Publication Date
524521-009	N.A.	February 2006
524521-010	N.A.	January 2007
524521-011	N.A.	August 2008
524521-012	N.A.	September 2008
524521-013	N.A.	March 2010
524521–014	N.A.	September 2010
524521–015	N.A.	October 2010
524521–016	N.A.	August 2011
524521–017	N.A.	February 2012
524521–018	N.A.	June 2012
524521–019	N.A.	February 2013
524521–020	N.A.	March 2014

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to:

pubs.comments@hp.com

Include the document title, part number, and any comment, error found, or suggestion for improvement you have concerning this document.

Request for Comments (RFC) Statement

This document uses information derived from RFC 2553, *Basic Socket Interface Extensions for IPv6*. The following copyright statement, copied from RFC 2553, is included in compliance with RFC 2553 copyright specifications:

Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

1 Introduction to Programming to the Guardian Sockets Library

This section discusses topics relating to sockets programming in the Guardian environment, including:

NonStop TCP/IP Subsystems and the Guardian Sockets Application Program Interface (API)

This manual documents the Guardian sockets API for the following four NonStop TCP/IP subsystems:

- NonStop TCP/IP (also called conventional TCP/IP)
- Parallel Library TCP/IP

  NOTE: Parallel Library TCP/IP is only supported on NonStop S-series servers.

- NonStop TCP/IPv6
- Cluster I/O Protocols (CIP)

Parallel Library TCP/IP and NonStop TCP/IPv6 share the same architecture; however, their architectures differ from that of conventional NonStop TCP/IP. For the most part, the different subsystem architectures do not affect the sockets API, with some exceptions. (See Multiple NonStop TCP/IP Processes and Logical Network Partitioning (LNP) (NonStop TCP/IPv6, H-Series and G06.22 and Later G-Series RVUs Only) (page 43)). For a comparison of the architectures of the three subsystems, see the *TCP/IPv6 Configuration and Management Manual*.

The greater difference, from a program-interface standpoint, lies in the difference between support for Internet Protocol version 4 (IPv4) and IPv6. NonStop TCP/IPv6 is the only NonStop TCP/IP subsystem that supports IPv6 communications. Writing and porting applications for IPv6 is discussed in Chapter 2. Where structures, header files, and library routines apply only to IPv6 and, therefore, only to the NonStop TCP/IPv6 product, this restriction is indicated in the text.

NonStop TCP/IPv6 has three operating modes: INET, INET6, and DUAL. When NonStop TCP/IPv6 runs in INET mode, it supports only IPv4 communications. In this mode, NonStop TCP/IPv6 is similar to Parallel Library TCP/IP and can be used instead of Parallel Library TCP/IP to achieve the same architectural advantages without the need to use the IPv6 capabilities. NonStop TCP/IPv6 continues to be enhanced and contains new features not available in Parallel Library TCP/IP, such as logical network partitioning. For this reason, your network administrator might have chosen to install the NonStop TCP/IPv6 subsystem instead of Parallel Library TCP/IP. If so, you can use NonStop TCP/IPv6 in INET or DUAL mode without any changes to your sockets applications. (In DUAL mode, if you do not change your application to support IPv6 addresses, your application can use the IPv4 addresses supplied by the subsystem.)

Parallel Library TCP/IP and NonStop TCP/IPv6 can coexist with conventional NonStop TCP/IP on the same system but not with each other.

CIP can coexist with NonStop TCP/IPv6 and conventional NonStop TCP/IP on the same system but not with Parallel Library TCP/IP since Parallel Library TCP/IP is not supported on J-series RVUs. CIP also supports IPv6.

CIP architecture differs from that of NonStop TCP/IPv6 and conventional NonStop TCP/IP; these differences affect the sockets API. For details about the CIP architecture and application compatibility, see the *Cluster I/O Protocols (CIP) Configuration and Management Manual*.

NOTE: Parallel Library TCP/IP is only available on NonStop S-series servers.

For information about transport-service provider names, see Process Names (page 43).

TCP/IP Programming Fundamentals

This subsection defines basic TCP/IP programming terms, concepts, and procedures:

Using NonStop TCP/IP and NonStop TCP/IPv6 or Parallel Library TCP/IP

An application process can have sockets associated with the NonStop TCP/IP, NonStop TCP/IPv6, and CIP environments; or the Parallel Library TCP/IP environment.

NOTE: Parallel Library TCP/IP is only available on G-series RVUs.

Using CIP

Applications that use the NonStop TCP/IP, Parallel Library TCP/IP, or TCP/IPv6 API might be affected by behavioral differences in the CIP API. For details on these differences, see the *Cluster I/O Protocols Configuration and Management Manual*. If you determine that these differences do not cause serious problems for your application, you can use an error suppression feature to allow the application to continue running if minor differences in the CIP environment are detected. This feature is described in the following subsection.

Suppressing Compatibility Errors

If you run an application in CIP that contains features that CIP does not support, compatibility errors result. To allow applications not expecting these errors to run without modification, CIP provides a DEFINE to suppress errors caused by incompatibility:

```
ADD DEFINE =CIP^COMPAT^ERROR, FILE SUPPRESS
```

If this DEFINE is set when an application starts, socket calls that result in a behavior allowed in a previous implementation, but not in CIP, return as if successful, even though the behavior did not occur as expected. If the DEFINE is not set or if the file name is not `SUPPRESS`, behaviors that CIP does not support cause socket calls to return an error.

Types of Service

Depending on the type of communications service required, your application uses one or more of the following protocols:

- The Transmission Control Protocol (TCP) provides reliable end-to-end data transfer. TCP is a stream-oriented protocol that has no concept of packet boundaries. TCP guarantees that all data sent is received and that the data arrives in the same order in which it was sent.
- The User Datagram Protocol (UDP) provides unreliable datagram service. The integrity of the packets sent is maintained; that is, when a packet is received, it matches exactly what was sent. However, neither the delivery of the datagrams nor the order in which the datagrams are received is guaranteed.
- The Internet Protocol (IP) allows data to be transferred between heterogeneous networks. It also services various host-to-host protocols. IP provides many capabilities at the network level and is the foundation of the NonStop TCP/IP subsystems. TCP and UDP use the Internet Protocol (IP). In addition, applications can provide their own Transport Layer protocols, built directly on IP.

The Socket Library Routines

All NonStop TCP/IP subsystems provide a socket interface that uses the HP NonStop operating system file-system procedures for interprocess communication and that provides socket library routines for the integration of UNIX and NonStop systems. You can use the socket library routines to access the socket interface programmatically.

A **socket** is an end point for communication. An application process calls a socket routine to request that the TCP/IP subsystem create a socket when needed and specify the type of service desired. Applications can request TCP and UDP sockets, as well as raw sockets, for direct access to the IP. (A **raw socket** allows direct access to a lower-level protocol.) The TCP/IP subsystem returns a socket number, which the application uses to reference the new socket.

After creating a socket, the application optionally binds the socket to a specific local address and port, and sends or receives data on the socket. When the transfer is complete, the application can shut down the socket and close it.

The NonStop server socket interface is modeled after the Berkeley Software Distribution (BSD) sockets interface to allow you to port existing UNIX TCP/IP applications to run on a NonStop system. For a description of the available socket-library routines, see Chapter 4 (page 81). For a summary of the differences between the NonStop TCP/IP socket interface and the 4.3 BSD UNIX interface, see Programming Using the Guardian Sockets Interface.

Although the NonStop server socket-library routines are based on the sockets programmatic-interface primitives in the 4.3 BSD release of the UNIX operating system, the NonStop server routines do not map exactly to the 4.3 BSD release function calls or functionality. The NonStop server routines include extensions to adapt the Berkeley sockets interface to HP fault-tolerant, operating-system features such as nowait I/O.

Beginning with the D30 RVU of NonStop TCP/IP, the socket library supports HP fault-tolerant applications (process pairs) written in either the C or TAL languages. This support is provided by two socket-library routines that permit the opening of sockets by a backup application. These routines are described in Chapter 4 (page 81) of this manual.

Servers and Clients

The terms **server** and **client** are used in the NonStop TCP/IP subsystems as they are customarily used in TCP/IP documentation. A server is a process that offers a service that can be used over the network; a server accepts requests, performs the specified services, and returns the results to requesters. A client is one of the processes that sends requests to the server and waits for it to respond. The client-server model is the same model known in other HP documentation as the

requester-server model—that is, a client is the same as a requester. Programming Using the Guardian Sockets Interface (page 32), explains how to develop client and server programs that use sockets.

Stream-Oriented Protocol Considerations

Unlike a protocol that sends and receives blocks or buffers of packets at a time, TCP is a stream-oriented protocol. The data has no boundaries except those put there by applications using TCP/IP. For example, the fact that the application sent 1,000 bytes does not mean the receiving end receives 1,000 bytes. The receiving end may only receive one byte; the network may only deliver in small chunks. The act of sending simply buffers the data for transfer, it does not imply that data has been sent or received. Completion of a receive simply provides the data that has been correctly received up to that point, up to the amount requested by the receive. When the application issues a receive function, all it specifies is how much data it can receive, that is, how big the buffer is. The application may get less data than it can receive.

If your application must be able to examine a whole record or block of data, it must embed data that marks or describes the blocks in the data. On the receiving end, the application receives the stream and looks for the block or record marks or has a previous definition of the record size. That is, if the application had a fixed record size of 80 bytes, the application would have to fragment the data itself. For example, if your application posted a receive for 1,000 bytes and received 800 (10 records X 80 bytes) the application would not need to fragment the data. But if the application posted a receive for 1,000 bytes and received 850 bytes, the application would have 10 whole records and one partial record and would need to keep track of the partial record, posting more receives to get the remaining data. The application also needs to know when it is finished, either through loss of connection, a pattern of bytes in the stream, a particular record type, or from some other event.

Passive Connect Compared to Active Connect

Passive connect means that the application sits listening for incoming connections, that is, passive connect posts an `accept` call. (In the OSS socket programming model, you would post a `listen` call.)

A server would most likely use the passive connect model.

The active connect model means the application initiates a connect by calling `connect` (or `connect_nw`). This call makes a connection to somebody listening for connections. Servers typically listen for connections.

Domain Name Resolution

When your program requests information about a host, the Domain Name resolver provides name-address resolution services. The Domain Name resolver is a programmatic interface consisting of socket-library support routines that get information about hosts, networks, protocols, and services. See Table 12 (page 82) for a list of these routines.

Depending on which support routine your program calls and the value defined for `=TCPIP^HOST^FILE` at the time the program runs, the Domain Name resolver accesses either a name server or one or both of two special host files that contain a list of Internet addresses and each of the corresponding hostname and alias(es) for those addresses. The default names of these files are `$SYSTEM.ZTCPIP.HOSTS` and `$SYSTEM.ZTCPIP.IPNODES`. (IPNODES is available for NonStop TCP/IPv6 or CIP.) If the address information is contained in some other file, each user running the program must define a value for `=TCPIP^HOST^FILE` and, for NonStop TCP/IPv6 or CIP, `=TCPIP^NODE^FILE`. Add DEFINE for `=TCPIP^NODE^FILE`, only when you want to place the IPNODES file in a location other than the default `$SYSTEM.ZTCPIP`.

The socket library uses the DEFINE command to resolve file names or process names. The DEFINE command is described in the *TACL Reference Manual*. Information about using the DEFINE command is in the *HP NonStop Kernel Operating System Users Guide*.

Also, see Using the DEFINE Command (page 29) for more information about setting file names and process names.

Your program calls `gethostbyname` and `getaddrinfo` routines to get the hostname and IP addresses. Guardian socket library gets the hostname and IP addresses as follows:

1. If there is a DEFINE for `=TCPIP^HOST^FILE`, and if hostname is found, it is returned from this file.

   If `=TCPIP^HOST^FILE` is not defined, DNS is queried for the hostname. If hostname is found, it is returned.

   If hostname is not found in DNS, default hosts file `$SYSTEM.ZTCPIP.HOSTS` is searched, and if found, hostname is returned.

   If hostname is not found in hosts file, HOST_NOT_FOUND error is returned in h_errno parameter.

2. If there is a DEFINE for `=TCPIP^NODE^FILE`, IP addresses for the given host are searched, and if IP addresses are found, they are returned.

   If host is not found in `=TCPIP^NODE^FILE`, and `=TCPIP^HOST^FILE` is defined, IP addresses are searched for in this file. If found, IP addresses are returned from the hosts file.

   If `=TCPIP^HOST^FILE` is not defined, Guardian socket library queries DNS for hostname.

   NOTE: Define `=TCPIP^HOST^FILE` to avoid querying DNS for IP addresses.

You can override the Guardian socket library's default behavior for hostname search by using PARAM, as shown below:

```
PARAM TCPIP^RESOLVER^ORDER value
```

where *value* is one of

DNSONLY
: Guardian socket library queries only DNS for the hostname.

HOSTFILEONLY
: Guardian socket library searches only the hosts file for hostname.

DNS-HOSTFILE
: Guardian socket library queries DNS. If hostname is not found, searches the hosts file for hostname.

HOSTFILE-DNS
: Guardian socket library searches the hosts file. If hostname is not found, it queries DNS for hostname.

NOTE: PARAM name and value are not case sensitive.

When the process has no PARAMs and DEFINEs, Guardian socket library queries the DNS for hostname.

Resolving Names With a Name Server

If a name server is available on the network, the recommended method for resolving names is to access the name server. To ensure that the resolver accesses a name server rather than a host file, your program should call the `gethostbyname` or `gethostbyaddr` routine or `getaddrinfo` or `getnameinfo` (for NonStop TCP/IPv6 or CIP), and program users should not define a value for `=TCPIP^HOST^FILE`.

To access a name server, the resolver uses information specified in a resolver configuration file. The default name for this file is `$SYSTEM.ZTCPIP.RESCONF`. (For a description of this file, see the *TCP/IPv6 Configuration and Management Manual* or the *Cluster I/O Protocols Configuration and Management Manual*.)

The NonStop server socket library uses the DEFINE command to resolve the file names and process names used by the socket library. See Using the DEFINE Command (page 29), for more information about the DEFINE command.

When a program sends a name-resolution request to the resolver, the resolver tries to send the query to the servers listed in the `RESCONF` file, sending the request to the server that has the highest priority first. The priority of a server depends on its position in the `RESCONF` file; the server listed first, called the *`primary server`*, has the highest priority. The `RESCONF` file can contain a maximum of 16 servers but must contain at least one server.

The resolver sends the request to the primary server using TCP port 53. If the primary name server does not respond within 4 seconds, the resolver tries to access the secondary name server; if that server does not respond within 4 seconds, the resolver tries to access the tertiary name server.

If none of the name servers responds within 4 seconds, the resolver retries the primary name server; however, this time the resolver waits up to 8 seconds for a response. If the primary name server does not respond within 8 seconds, the resolver tries the secondary name server. If that server does not respond within 8 seconds, the resolver tries the tertiary name server.

The resolver continues trying to access each name server, increasing the time it waits for a response, from 4 to 8 to 16 and then to 32 seconds in each of the subsequent retry cycles. Failure conditions are stored in the external variable *`h_errno`*. The errors returned in *`h_errno`* are described along with the `gethostbyaddr` and `gethostbyname` functions in Chapter 4 (page 81).

If the name server cannot be accessed (that is, does not respond to requests), the HOSTS-type file is accessed in an attempt to resolve the name. If the name server can be accessed but cannot resolve the name, the resolver routine returns an error and the HOSTS-type file is not checked.

NOTE: Beginning with the D40.00 RVU of NonStop TCP/IP, the socket-library routine gethostbyname() was changed with respect to name server lookups. If the name server cannot resolve the name, or the name server does not respond, the HOSTS-type file is accessed.

Resolving Names by Using a HOSTS-Type File

If a name server is not available on the network, you can resolve names by using a HOSTS-type file. This nonstandard technique for resolving names can be implemented using either of two methods:

- From a program, call one of the following routines:
  - `host_file_gethostbyname`
  - `host_file_gethostbyaddr`

    Defining a value for `=TCPIP^HOST^FILE` is optional for this method. The only reason for defining a value for `=TCPIP^HOST^FILE` is to specify a file other than the default file to resolve names.
- From a program, call one of the following routines:
  - `gethostbyname`
  - `gethostbyaddr`
  - `getaddrinfo` (NonStop TCP/IPv6)
  - `getnameinfo` (NonStop TCP/IPv6)

    With this method, users running the program must define a value for `=TCPIP^HOST^FILE` before running the program.

With either method, TCP/IP resolves the names by using either the `$SYSTEM.ZTCPIP.HOSTS`, the `$SYSTEM.ZTCPIP.IPNODES` (for NonStop TCP/IPv6 and CIP) file or a file name specified

in a previous ADD DEFINE command that defines a value for =TCPIP^HOST^FILE or =TCPIP^NODE^FILE.

The socket library uses the DEFINE command to resolve the file names and process names used by the socket library. For more information, see Using the DEFINE Command (page 29).

ND6HOSTD Process for NonStop TCP/IPv6

The ND6HOSTD process for NonStop TCP/IPv6 is a utility process that you can run to receive and process router advertisement (RA) packets and update the global address information in the DNS. The ND6HOSTD process is a Guardian process started by the $ZPM persistence manager. It runs in one or more processors in which a TCP6MON is running. For more information about ND6HOSTD, see the *TCP/IPv6 Configuration and Management Manual*.

Starting Clients and Servers

Typically, a client program is started by an application user at a terminal. A server might be started by an operator or system manager, or by the LISTNER process, depending on the way you design and set up the server. When a client or server program is started, the person starting the program might need to set one or more TCP/IP attributes to control how the program operates.

NOTE: You should use the standard configuration, so that users running the client and server programs do not need to enter DEFINE commands. Use a nonstandard approach only when the normal one does not meet the needs of your application. However, if you are using CIP, you might want to set the compatibility error suppression DEFINE, as described under "Suppressing Compatibility Errors" (page 24). For descriptions of CIP compatibility considerations, see the *Cluster I/O Protocols (CIP) Configuration and Management Manual*. You can use this information to determine how your application might be affected by compatibility issues and whether or not to set the compatibility error suppression DEFINE.

Using the DEFINE Command

The socket library uses values defined by the ADD DEFINE command to resolve file names and process names as well as to provide some other functions for the library. The following DEFINE names affect the operation of NonStop TCP/IP, Parallel Library TCP/IP, NonStop TCP/IPv6, and CIP programs (both those provided by HP and the ones you develop):

=PTCPIP^FILTER^KEY	Defines the key or password for round-robin. (Parallel Library TCP/IP and NonStop TCP/IPv6 only)
=PTCPIP^FILTER^TCP^PORTS	Limits the TCP ports that applications share in round-robin filtering (Parallel Library TCP/IP and NonStop TCP/IPv6 only)
=PTCPIP^FILTER^UDP^PORTS	Limits the UDP ports that applications share in round-robin filtering (Parallel Library TCP/IP and NonStop TCP/IPv6 only)
=TCPIP^HOST^FILE	Specifies the name of the HOSTS-type file to be used to resolve names
=TCPIP^NODE^FILE	Specifies the name of the IPNODES file to be used to resolve names (NonStop TCP/IPv6 only)
=TCPIP^NETWORK^FILE	Specifies the network addresses and names for `getnetbyaddr` and `getnetbyname` functions
=TCPIP^PROTOCOL^FILE	Specifies protocol names and port numbers for `getprotobyname` and `getprotobynumber` functions
=TCPIP^RESOLVER^NAME	Specifies the name of the resolver configuration file to be used to get name server information
=TCPIP^SERVICE^FILE	Specifies service by port number and name for `getservbyname` and `getservbyport` functions
=_SRL_01	Defines the SRL for the TCPSAM process. (Parallel Library TCP/IP and pre-G06.24 RVU NonStop TCP/IPv6 only.)

`=TCPIP^PROCESS^NAME`	Specifies the name of the NonStop TCP/IP process or TCPSAM or TCP6SAM process name
`=CIP^COMPAT^ERROR,FILE SUPPRESS`	When set with a file name of "SUPPRESS", specifies that when an application starts, socket calls that try to invoke a behavior allowed in a previous implementation, but not in CIP, return as if successful even though the behavior did not occur as expected.

The runtime entries for various files should be:

```
ADD DEFINE =TCPIP^HOST^FILE, FILE $SYSTEM.ZTCPIP.HOSTS
ADD DEFINE =TCPIP^NODE^FILE, FILE $SYSTEM.ZTCPIP.IPNODES
ADD DEFINE =PTCPIP^FILTER^KEY, CLASS MAP, FILE file-name
ADD DEFINE =TCPIP^NETWORK^FILE, FILE $SYSTEM.ZTCPIP.NETWORKS
ADD DEFINE =PTCPIP^FILTER^TCP^PORTS, FILE Pstartport.Pendport
ADD DEFINE =PTCPIP^FILTER^UDP^PORTS, FILE Pstartport.Pendport
ADD DEFINE =TCPIP^PROTOCOL^FILE, FILE $SYSTEM.ZTCPIP.PROTOCOL
ADD DEFINE =TCPIP^RESOLVER^NAME, FILE $SYSTEM.ZTCPIP.RESCONF
ADD DEFINE =TCPIP^SERVICE^FILE, FILE $SYSTEM.ZTCPIP.SERVICES
ADD DEFINE =_SRL_01, CLASS MAP, FILE ZTCPSRL
ADD DEFINE =TCPIP^PROCESS^NAME, FILE $ZTC0
ADD DEFINE =CIP^COMPAT^ERROR, FILE SUPPRESS
```

A value for `=TCPIP^PROCESS^NAME` must be defined only if both the following conditions exist:

- The transport-service-provider process on your system has been configured with a name other than $ZTC0.
- The program that is going to be run does not call the `socket_set_inet_name` routine to specify a NonStop TCP/IP, TCPSAM, TCP6SAM, or CIP process name. A call to this routine overrides both the default name $ZTC0 and `=TCPIP^PROCESS^NAME` (if it is defined).

A value for `=TCPIP^RESOLVER^NAME` must be defined only if both the following conditions exist:

- The program that is going to be run calls the `gethostbyname`, `gethostbyaddr`, `getnameinfo`, or `getaddrinfo` routines.
- The name-server information normally contained in the `$SYSTEM.ZTCPIP.RESCONF` file is contained in some other file.

For a DEFINE name to be available to a program, the DEFINE name must be defined prior to running the program. When you define a DEFINE name during an interactive session at a terminal, the DEFINE name stays in effect until you clear it (using the DELETE DEFINE command), redefine it through another ADD DEFINE command, or log off from the session. You can also use the SHOW DEFINE command to list DEFINE name values you have defined. The attributes of an established DEFINE name can be changed using the ALTER DEFINE command. Descriptions of the various DEFINE commands appear in the *TACL Reference Manual*.

The following example shows you how to use the ADD DEFINE command to set up the host file. Here, `$TESTV.TSUBV.HOSTXX` is defined to be the file used for resolving domain names. Then, a server program named XXTEST (which uses the HOSTXX file to resolve domain names) is run:

```
TACL 3> ADD DEFINE =TCPIP^HOST^FILE,FILE $TESTV.TSUBV.HOSTXX
.
.
.
TACL 4> RUN XXTEST
```

Always specify a fully qualified file name for the `=TCPIP^HOST^FILE` value.

If your system has been configured to have a TCP/IP process named $ZTCM, you must define `=TCPIP^PROCESS^NAME` before running any clients or servers that use the TCP/IP subsystem (the operator or system manager who starts the NonStop TCP/IP, Parallel Library TCP/IP, NonStop TCP/IPv6, or CIP process must also define `=TCPIP^PROCESS^NAME`):

```
TACL 5> ADD DEFINE =TCPIP^PROCESS^NAME, FILE $ZTCM
```

LISTNER Process

The LISTNER process functions as a "super server" for some application servers provided by HP (such as the FTP server). LISTNER invokes the appropriate NonStop server as connection requests for services are received on well-known TCP ports (in the default configuration). These services do not apply to UDP ports. The use of a single super server—in this case, the LISTNER process—to invoke several other servers, effectively reduces the load on the system.

To use the LISTNER process, you must configure the `PORTCONF` file and start the LISTNER process. The `PORTCONF` file defines the servers to be invoked when a request comes in from another system on the Internet. Once started, LISTNER reads the `SERVICES` file to resolve the services configured in the `PORTCONF` file. (The `SERVICES` file is provided with the NonStop TCP/IP, Parallel Library TCP/IP, NonStop TCP/IPv6, and CIP software.) LISTNER checks that the service name and corresponding port are valid.

You can configure the `SERVICES` and `PORTCONF` files using port numbers other than the well-known port numbers for the services. For information about configuring and starting the LISTNER process, see the *TCP/IP Applications and Utilities User Guide.*

Once the accuracy of the `PORTCONF` file contents is verified by using the `SERVICES` file, LISTNER "listens" to the configured ports that are waiting for incoming connection requests from the remote client. The TCP/IP process notifies the LISTNER process when a request is pending.

When the LISTNER process receives the notification, it starts the server targeted by the request. The target server creates a socket using host-name and source-port information, then accepts the pending connection request on the newly created socket.

Data can be transferred between the NonStop target server and the remote client through the newly created socket until either the remote client or the target server terminates the connection.

Port Numbers

Both TCP and UDP use a 16-bit port number to select a socket on the host. Client programs normally use more or less random port numbers; however, specific port numbers—called well-known ports—are assigned for use by server programs.

Each well-known port is associated with a specific service. A client requesting a particular service (such as file transfer) specifies as the destination port the port associated with that particular service. The server program monitors that port for file-transfer requests. The well-known port numbers for TCP and UDP are listed in Appendix A (page 241) in this manual.

In TCP, the combined remote IP address, remote port number, local IP address, and local port number uniquely identify a connection. In UDP, the same four parameters identify a temporary source and destination. These four parameters are part of every TCP or UDP packet that passes over the Internet.

Each separate session must have a unique combination of these four parameters. However, any three of the parameters can be the same as long as the fourth is different. For instance, two different applications on the same host can send files at the same time to another host, which can also be the same, as follows:

```
            IP Addresses              Port Numbers
            (source, destination)     (source, destination)

Session 1   122.1.7.19, 101.3.5.2     1281, 21
Session 2   122.1.7.19, 101.3.5.2     1282, 21
```

Because the same host systems are involved, the IP addresses are the same. Because both sessions are file transfers, one end of both sessions involves the well-known FTP port number 21 (for the file-transfer service). The only difference in the two sessions lies in the port numbers for the applications requesting the service.

Generally, at least one end of the session requests a port number that is guaranteed to be unique. The client program normally requests the unique port number, because the server typically uses a well-known port.

Network and Host Order

In the descriptions of some of the support routines in the socket library, this manual refers to IP addresses or port numbers as being in network order or in host order. These terms refer to the routines the order in which the octets are stored in arguments passed to or returned by the routines. On NonStop operating systems, network order is the same as host order.

The Internet standard for the transmission of 32-bit integers specifies that the most-significant octet should appear first. However, not all hosts store integers in the same way. Thus, copying octets directly from one host to another can change the value of a number. The Internet standard specifies that sending hosts must translate from their local integer representation (local order) to network order (most-significant octet first). Receiving hosts are required to translate from network order to local order.

Programming Using the Guardian Sockets Interface

This subsection provides guidelines for programming to the Guardian sockets library, including:

- Porting Considerations
- Nowait I/O (page 32)
- Differences Between UNIX and NonStop Server Implementations (page 33)
- NonStop TCP/IP, Parallel Library TCP/IP, and NonStop TCP/IPv6 Basic Steps (page 35)

Porting Considerations

The socket library routines are based on the 4.3 BSD implementation of the UNIX operating system. However, there are some differences, mostly resulting from differences between the NonStop operating system and the UNIX environment. Therefore, some parts of your programs need to change if you are porting them from the 4.3 BSD UNIX operating system or from some other TCP/IP implementation.

Nowait I/O

Nowait I/O in the NonStop operating-system environment is similar to nonblocking I/O in UNIX, but there are important differences. First, nowait I/O can be performed only over a socket that was created for nowait I/O (with a call to the `socket_nw` function). Once a socket is created, it cannot be switched from one mode to the other.

The following nonstandard socket calls are available for nowait I/O:

`accept_nw`	`getsockopt_nw`	`shutdown_nw`
`accept_nw1`	`recv_nw`	`socket_nw`
`accept_nw2`	`recvfrom_nw`	`t_recvfrom_nw`
`bind_nw`	`send_nw*`	`t_sendto_nw`
`connect_nw`	`send_nw2`	`t_sendto_nw64_`
`getpeername_nw`	`sendto_nw`	`sendto_nw64_`
`getsockname_nw`	`sendto_nw64_`	`sendto_nw2_64_`
`recvfrom_nw64_`	`setsockopt_nw`	`recv_nw64_`
`send_nw2_64_`	`t_recvfrom_nw64_`	

In most cases, the parameters for these calls are identical to those of the corresponding waited calls, with the addition of extra parameters for NonStop operating system requirements. The

exceptions to this rule are `accept_nw2`, `recvfrom_nw`, `recvfrom_nw64_`,`send_nw2`, `send_nw2_64_`, `sendto_nw`, `sendto_nw64_`, `t_recvfrom_nw`, `t_recvfrom_nw64_`, `t_sendto_nw` and `t_sendto_nw64_`, which have different sets of parameters.

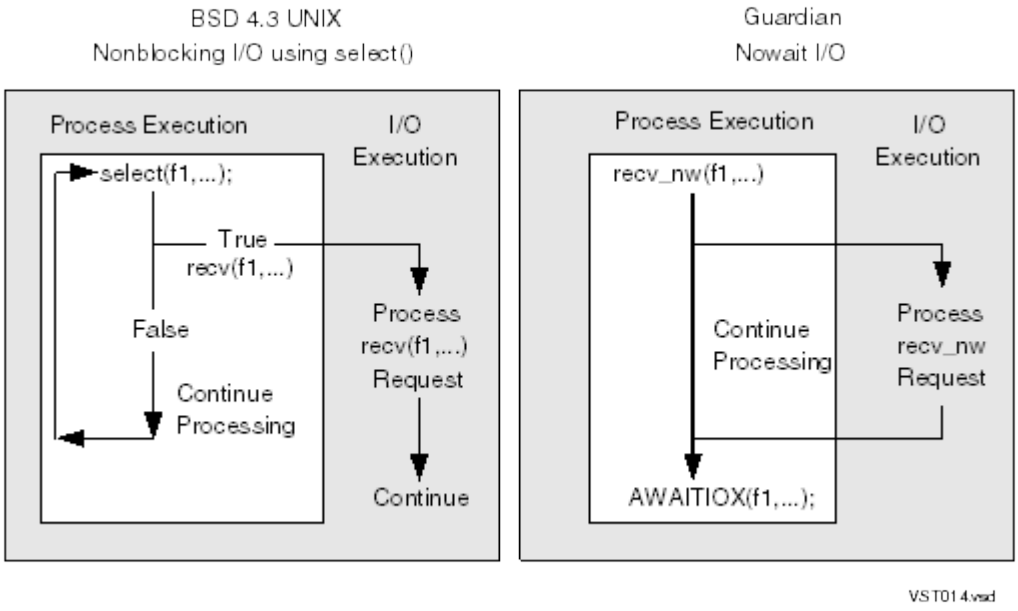
In addition, a nowait I/O operation is never performed synchronously, and the error EWOULDBLOCK is never returned. After performing a nowait I/O operation, your program must check for completion by issuing a call to the `AWAITIOX` or `FILE_AWAITIO64_` procedure call.

The examples in Figure 1 (page 33) summarize the procedural differences between 4.3 BSD UNIX nonblocking I/O and NonStop operating system nowait I/O.

In 4.3 BSD UNIX, the application tests (polls) a socket (f1) by using the select call check whether I/O activity, in this case receiving data, can occur on the socket. If the socket can receive data, the application issues the recv call; otherwise, the application continues processing, then again issues the select call to poll the socket.

In the NonStop operating-system environment, the application issues the `recv_nw` call on a socket (f1) to attempt to receive data on a socket. The application continues processing, then calls `AWAITIOX` to determine if the `recv_nw` call has completed.

Figure 1 4.3 BSD UNIX Nonblocking I/O Compared to Guardian Nowait I/O



Differences Between UNIX and NonStop Server Implementations

The NonStop server socket routines also differ from the 4.3 BSD UNIX socket routines in the following ways:

- The `select` routine is not supported. Instead, use the nowait I/O capability to test I/O completion by issuing the `AWAITIO[X]` call on specific sockets.
- Include files are in the $SYSTEM.ZTCPIP subvolume, rather than in the /usr/include directory.
- The NonStop operating system does not have a facility corresponding to UNIX signals. Therefore, the NonStop TCP/IP, Parallel Library TCP/IP, and NonStop TCP/IPv6 software returns the error `EHAVEOOB` to indicate that urgent (out-of-band) data is pending. Whenever this error occurs, your program must clear the out-of-band data before proceeding, by calling either `recv` or `recv_nw` with *`flags`* set to `MSG_OOB`.
- The I/O Control operations available for sockets are restricted. Although most of the socket I/O Control operations are available, `SIOCGIFCONF` and `FIONBIO` are not supported. Those I/O Control operations available are accessed through the `socket_ioctl` function. For a complete list of the I/O Control operations supported, see Table 16 (page 199).

- Because of differences between the UNIX and NonStop operating system I/O environments, some differences exist in the errors returned in *errno* by the socket routines. Although errors that have the same names are compatible, some error numbers do not match those returned by UNIX implementations. Programs that refer to errors by number rather than by name require a greater conversion effort.

  In particular, those socket errors that represent UNIX operating-system-dependent errors are not returned, and NonStop operating system file-system errors can be returned. For details, see Appendix B.

- Sockets can be closed or removed only by calling the file-system procedures `FILE_CLOSE` or `CLOSE`.
- File control provided by the UNIX `fcntl` system call is not supported.
- The functions `recv[from]_nw` and `t_recvfrom_nw` require the use of the `AWAITIOX` procedure to determine the number of characters read.
- The function `send[to]_nw` requires the use of the `AWAITIOX` procedure to determine the number of characters sent. If the amount of data sent is less than the length of the message, issue another pair of `send_nw` and `AWAITIOX` calls.

  To determine the number of characters sent through a call to `send_nw` or `t_sendto_nw`, you can alternatively look at *nb_sent*, which is the first parameter of `struct send_nw_str`. See the description of the `send_nw` routine in Chapter 4 for information about this structure.

- The NonStop server implementation of database-support routines such as `gethostbyaddr`, `gethostbyname`, `getnameinfo`, and `getaddrinfo`, are all waited calls.
- NonStop TCP/IP, Parallel Library TCP/IP, and NonStop TCP/IPv6 sockets provide the `sockaddr` data structure for IP address, address family, and port information as a pointer to the HP-defined `sockaddr_in` data structure. Functionality for both data structures is identical.
- In the NonStop TCP/IP, Parallel Library TCP/IP, and NonStop TCP/IPv6 implementations, Read and Write operations are not supported for Guardian sockets.

Asynchrony and Nowaited Operations

Asychrony mechanisms differ depending on whether you are dealing with OSS sockets or Guardian sockets (for OSS, see the *Open System Services Programmer's Guide*). Asynchrony refers to the issuance and completion of an operation occurring at different times. Synchronous operations happen stepwise when your program runs; that is, the completion occurs as a result of returning from the function.

In Guardian, specific versions of the library routines (functions) end in `_nw`; for example, `send_nw` and `recv_nw`. `nw` stands for nowait. (See Nowait I/O (page 32) for more information.)

A function is initiated upon return of the function call but the function is not necessarily completed. At some point, for a Guardian program, the application runs out of things to do and is ready to wait for notification about completion of all the different asynchronous functions that the application has initiated. This behavior is typical of servers. Servers cannot afford to wait for operations to complete because waiting means they are not serving someone else. Eventually, the server calls AWAITIOX which is a Guardian function that allows the application to rendezvous and either wait or get any completions that are pending. If no functions are finished, AWAITIOX waits as long as you specified in a parameter that you sent to AWAITIOX. This wait time can be anywhere from 0 to infinity. Eventually, when the completion occurs, AWAITIOX returns and tells the application why it woke up (AWAITIOX can wait for multiple reasons.)

When the application gets a return from AWAITIOX, the parameter returned is a file number which corresponds to the socket. A tag is also returned. One parameter to `recv_nw`, `send_nw`, is a tag, because if the application is doing multiple operations at once, it must be able to differentiate between the operations. So a unique value is associated with each operation (for example, multiple

sends on the same socket). AWAITIOX returns a tag and a socket ID so the application can identify which operation just completed. At that point, the application issues a `FILE_GETINFO_` call using that file number to get back the completion status of the operation the application just performed (and any other fields such as return length, depending on the operation).

Considerations for Using socket_nw

If you have a server which cannot afford to wait, rather than using the socket call, you should use `socket_nw`. Similarly, if your server cannot afford to wait, use `send_nw`.

Concurrency and Considerations for Blocking and Nonblocking

Asynchrony is a way an application can achieve concurrency of your server's execution with the execution of the TCP/IP protocol. By using asynchronous operations, you ensure the concurrent execution of your program with the completion of the work done by the TCP/IP protocol stack.

In OSS, mechanisms for asynchrony are similar to but distinct from the Guardian mechanisms for asynchrony. The OSS mechanism is derived from the UNIX world, where instead of waited and nowaited operations, you have the notion of blocking and nonblocking operations. Blocking operations are similar to Guardian waited operations. Control does not return back to your program until the operation has completed.

Nonblocking means that the application can issue an operation as nonblocking and the application can get the completion of the operation later. This way, the operation proceeds concurrently with your application's operation. (See Nowait I/O (page 32) for a more in-depth comparison of waited and nowaited operations compared to blocking and nonblocking operations.)

NOTE: A receive must be posted on a socket for the data to be acted on.Your application should post the receive before the send is issued so there is no time lag.

Considerations for a Server Posting Receives

From a system standpoint, a server should post the biggest receives it can consistent with the maximum size of what the other can send. The larger the receive the server can post, the better. If the other side has control over how much can be sent, the more sent the better. A server should have at least one receive pending on every socket on which it can simultaneously receive data. Because TCP is a streaming protocol, you might want to have more than one receive pending on any socket because you may get data coming in a little at a time. More importantly, you want to ensure a large enough receive-space parameter by setting a socket option (`SO_RCVBUF`).

Basic Steps for Programs

This subsection summarizes the basic steps performed by a client and server program for the NonStop TCP/IP, Parallel Library TCP/IP, and NonStop TCP/IPv6 subsystems.

NonStop TCP/IP, Parallel Library TCP/IP, and NonStop TCP/IPv6 Basic Steps

The basic steps performed by a client or server program are the same whether your program uses TCP sockets, UDP sockets, or RAW sockets. This subsection summarizes these steps for each type of program. Important considerations for each type of program are presented later in this section.

Client Program

The basic steps performed by a client program are:

1. Designate the NonStop TCP/IP, Parallel Library TCP/IP, or TCP6SAM process name (optional).
2. Create a socket.
3. Bind the socket to any port (optional; not done for RAW).
4. Connect the socket (required for TCP; optional for UDP and RAW).
5. Start data transfer.

6. Shut down the socket (optional for TCP; not done for UDP or RAW).
7. Close the socket.

Designating the NonStop TCP/IP, TCPSAM, or TCP6SAM Process Name

To create a socket, the socket-interface library opens a file to communicate with the NonStop TCP/IP, TCPSAM, or TCP6SAM process. Therefore, the socket library must know the name of this process before any sockets are created. Programs can specify this process explicitly by calling the function `socket_set_inet_name`.

If a program has not called `socket_set_inet_name` before creating a socket, the function that creates a socket makes default assumptions about the process name. The function uses the value of `TCPIP^PROCESS^NAME`, if it exists (usually declared using the DEFINE command); otherwise, it uses the process name $ZTC0. See Using the DEFINE Command (page 29), for more information about the value of `TCPIP^PROCESS^NAME`.

Creating a Socket

A program calls the socket function to create a socket. The socket function returns a descriptor. The program passes this socket descriptor to subsequent calls for operations on that socket.

Binding a Socket

A program can associate the socket with a local address and port number by calling the `bind` function. This call is optional for client programs. If the program does not call `bind`, the `connect` function performs the binding.

For UDP and RAW, calls to `bind` and `connect` are unnecessary because UDP and RAW datagrams contain all the addressing information needed. UDP datagrams contain information about source and destination addresses and port numbers. RAW datagrams contain information about source and destination addresses; however, unlike UDP, the RAW datagrams use protocol numbers instead of port numbers. You specify the protocol number in the `socket` call.

Connecting a Socket

The `connect` function associates a remote address and port number with the socket. For TCP, `connect` issues a request for an active connection. For UDP and RAW, no active connection exists; `connect` merely serves as a convenient means to permanently specify the remote address and port number (or protocol number) so that each call to transfer data does not need to specify this information. For UDP or RAW, your program can either call `connect` to specify the remote address and port/protocol number once, or the program can use the `sendto` or `recvfrom` routines.

Transferring Data

Two sets of routines are provided for sending and receiving data. One set, the `send` and `recv` routines, uses the remote address and port number specified for the socket in a previous call to `connect`. The other set, the `sendto` and `recvfrom` routines, uses the remote address and port number passed as an argument in the call. The `sendto` and `recvfrom` routines are provided for use with connectionless protocols (UDP and RAW) in programs that do not call `connect`.

Shutting Down and Closing a Socket

The `shutdown` routine shuts down data transfer on an actively connected TCP socket, either partially or completely (preventing further reads, writes, or both). Calling `shutdown` is optional; if a program does not call `shutdown`, a call to the `CLOSE` or `FILE_CLOSE_` procedure performs the shutdown procedure. Because `shutdown` applies to an active connection, a program using UDP sockets or raw sockets does not need to call this routine.

When communication is complete, your program must close the socket explicitly by issuing a call to `FILE_CLOSE_` or `CLOSE`, passing it the socket number as is done for the socket routine calls.

Server Program

The basic steps performed by a server program are:

1. Designate the NonStop TCP/IP, TCPSAM, or TCP6SAM process name (optional).
2. Create a socket.
3. Bind the socket to a well-known port (required for most servers; does not apply to RAW; optional for servers started by the LISTNER process).
4. Listen for connections (required for TCP; not done for UDP or RAW).
5. Accept incoming connections. When a connection is received, create a new socket and accept the connection on the new socket (required for TCP; optional for UDP; not done for RAW).
6. Start data transfer (if step 5 was done, use the new socket created in that step).
7. Shut down the socket (optional for TCP; not done for UDP or RAW).
8. Close the socket.

For servers, some of the calls or call requirements vary depending on the way the server operates. Servers that operate at a well-known port (one that is associated with a specific service provided by the server) must perform a call to `bind` to permanently associate the socket with that port.

Steps 1 through 3 and 6 through 8 are used in the same way by servers and clients. See TCP Client and Server Programs (page 39) for descriptions of the similar steps. The steps for listening for and accepting connections apply only to servers; these steps are described below.

Listening for Connections

The `listen` routine is provided in the 4.3 BSD UNIX operating system to set the queue length for pending TCP connections on a socket. The NonStop TCP/IP process or Parallel Library TCP/IP, or NonStop TCP/IPv6 subystem sets a default value of 5 for the queue length. Using the `listen` routine, you can set the queue length to a value from 1 through 5; TCP servers must call `listen` before accepting a connection.

Accepting a Connection

A server typically uses one socket to check for connections and another socket to transfer data (if the same process performs both functions). This technique allows the server to check for a new connection on the first socket, accept the new connection, and start data transfer on a second socket. The server can then check for another new connection on the first socket without waiting for the data transfer to complete. The `accept` routine permits this type of operation.

The `accept` routine performs three steps. First, the routine checks for connections on an existing socket. Then, when a connection request arrives, `accept` creates a new socket for the data transfer. Finally, it accepts the connection on the new socket. For nowait operations, a program must issue a sequence of these calls to perform these functions:

```
accept_nw
AWAITIOX
socket_nw
AWAITIOX
accept_nw2
AWAITIOX
```

Server Programs Started by LISTNER

The LISTNER process described in LISTNER Process (page 31), checks for connections. When LISTNER receives a connection request, it starts another process and passes the connection information to that process, which in turn handles the data transfer. The LISTNER process calls `accept_nw`. After the `AWAITIOX` command completes, LISTNER passes the returned remote address and port number to the second process.

If you are programming a server that you want LISTNER to start, your server program must call `socket` to create a socket, call `bind` to bind the socket to a local address and port, and then call `accept_nw2` to accept the connection for data transfer (passing to `accept_nw2` the socket

number of the socket created by your server program and the remote address and port number passed from LISTNER).

The programming example on the following pages uses LISTNER to start a server:

```
#include <socket.h>
#include <in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
int Accept_Conn(char*);
int sock = -1;
int main(int argc, char *argv[])
{
   int nrcvd;
   char buf[1024], *cp;
   /*
    * If this has been started by a server, then
    * accept a connection; otherwise, echo to
    * stdout from stdin.
    */
if (argv[1] != (char *)NULL) {
      /*
 * argv[1] must have port.hostname format.
       */
      if ((cp = strchr(argv [1], '.')) == (char *)NULL) {
         fprintf(stderr, "Server: bad arg %s\n", argv[1]);
         exit(1);
      }
      *cp = 0;
      if (atoi(argv[1]) == 0) {
         fprintf(stderr, "Server: bad arg %s\n", argv[1]);
         exit(1);
      }
      *cp = '.';
      if (Accept_Conn(argv[1]) == 0)
         exit(1);
   }
   if (sock >= 0)
      while ((nrcvd = recv(sock, buf, (int)sizeof(buf), 0)) > 0)
          send(sock, buf, nrcvd, 0);
   else
      while ((nrcvd = read(fileno(stdin), buf, (int)sizeof(buf))) > 0)
          write(fileno(stdout), buf, nrcvd);
   exit(0);
}

/* Accept an incoming connection request.
 * The argument passed to us in the form:
 *
 *   PORT.HOST
 */
memset (&sin, 0, sizeof(sin));
int Accept_Conn(char* cp)
{
   struct sockaddr_in sin;
   /*
    * Set up the sock_addr_in structure based on the
    * argument.
    */
   sin.sin_port = atoi (cp);
   cp = strchr (cp, '.') + 1;
   if ((sin.sin_addr.s_addr = inet_addr (cp)) == 0) {
      printf ("Bad value for %s\n", cp);
```

```
      return 0;
   }
   sin.sin_family = AF_INET;
   /*
    * Create a socket so that we can use it for
    * accepting the connection.
    */
   if ((sock = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
      perror ("socket");
      return 0;
   }
   /*
    * This is a waited socket, but we use the trick of
    * nowait accept_nw2, because this does just what we
    * need (accept a connection as a new socket).
    */
   if (accept_nw2(sock, (struct sockaddr*)&sin, 01) < 0) {
      perror ("accept_nw2");
      return 0;
   }
   return 1;
}
```

TCP Client and Server Programs

Table 1 lists the steps performed by a TCP client and a TCP server in waited operations. The calls used to perform each step are given in parentheses; calls spelled out in uppercase letters are NonStop operating system procedure calls.

Table 1 TCP—Waited Client and Server Steps

Client		Server	
1.	Optionally, set NonStop TCP/IP or TCP6SAM process name (`socket_set_inet_name`).	1.	Optionally, set NonStop TCP/IP or TCP6SAM process name (`socket_set_inet_name`).
2.	Create a socket (`socket`).	2.	Create a socket (`socket`).
3.	Optionally, bind the socket to any port (`bind`).	3.	Bind the socket to a well-known port (`bind`).
4.	Connect the socket to the server (`connect`).	4.	Listen for connections (`listen`).
		5.	Accept an incoming connection on a new socket (`accept`).
5.	Start data transfer (`send` and/or `recv`, usually in a loop).	6.	Start data transfer on the new socket (`recv` and/or `send`, usually in a loop).
6.	Optionally, shut down the socket (`shutdown`).	7.	Optionally, shut down one or both sockets (`shutdown`).
7.	Close the socket (`CLOSE or FILE_CLOSE_`).	8.	Close the socket (`CLOSE or FILE_CLOSE_`).

Table 2 (page 40) shows the steps performed by a TCP client and a TCP server in nowait operations. The calls used to perform each step are given in parentheses. Note the use of nowait versions of most of the socket calls, followed by calls to the `AWAITIOX` procedure for completion of the call.

The nowait versions of the socket calls require the program to provide a *`tag`* parameter to identify the particular operation. When `AWAITIOX` is called, it returns the *`tag`* that was passed to it in the corresponding nowait socket call.

Sample TCP client and server programs are provided in Chapter 5.

Table 2 TCP—Nowait Client and Server Steps

Client	Server
1. Optionally, set NonStop TCP/IP, TCPSAM, or TCP6SAM process name (`socket_set_inet_name`).	1. Optionally, set NonStop TCP/IP, TCPSAM, or TCP6SAM process name (`socket_set_inet_name`).
2. Create a socket (`socket_nw`, followed by `AWAITIOX`).	2. Create a socket (`socket_nw`, followed by `AWAITIOX`).
3. Optionally, bind the socket to any port (`bind_nw`, followed by `AWAITIOX`).	3. Bind the socket to a well-known port (`bind_nw`, followed by `AWAITIOX`).
4. Connect the socket to the server (`connect_nw`, followed by `AWAITIOX`).	4. Listen for connections (`listen`).
	5. Accept the connection. a. Accept an incoming connection (`accept_nw`, followed by `AWAITIOX`). b. Create a new socket (`socket_nw`) with (`flags & 0200`) nowait set.c. Call `AWAITIOX`, followed by `SETMODE 30`, followed by `AWAITIOX`). d. Accept the new connection on the new socket (`accept_nw2`, followed by `AWAITIOX`).
5. Start data transfer (`send_nw` and/or `recv_nw`, followed by `AWAITIOX`, usually in a loop).	6. Start data transfer on the new socket (`recv_nw` and/or `send_nw`, followed by `AWAITIOX`, usually in a loop).
6. Optionally, shut down the socket (`shutdown_nw`, followed by `AWAITIOX`).	7. Optionally, shut down one or both sockets (`shutdown_nw`, followed by `AWAITIOX`).
7. Close the socket (`CLOSE or FILE_CLOSE_`).	8. Close the socket (`CLOSE or FILE_CLOSE_`).

UDP Client and Server Programs

Table 3 shows the steps performed by a UDP client and a UDP server in waited operations.

Table 3 UDP—Waited Client and Server Steps

Client	Server
1. Optionally, set NonStop TCP/IP, TCPSAM, or TCP6SAM process name (`socket_set_inet_name`).	1. Optionally, set NonStop TCP/IP, TCPSAM, or TCP6SAM process name (`socket_set_inet_name`).
2. Create a socket (`socket`).	2. Create a socket (`socket`).
3. Optionally, bind the socket to any port (`bind`).	3. Bind the socket to a well-known port (`bind`).
4. Start data transfer (`sendto` and/or `recvfrom`, usually in a loop). OR Specify the remote address for the socket (`connect`). Then, start data transfer (`send` and/or `recv`, usually in a loop).	4. Start data transfer (`recvfrom` and/or `sendto`, usually in a loop). OR Specify the remote address for the socket (`connect`). Then, start data transfer on the socket (`recv` and/or `send`, usually in a loop).
5. Close the socket (`CLOSE or FILE_CLOSE_`).	5. Close the socket (`CLOSE or FILE_CLOSE_`).

See Usage/Bind Considerations (page 87) for information about the HP implementation that handles the binding of UDP sockets. The implementation ensures that the correct process is notified when a broadcast message arrives.

Table 4 shows the steps performed by a UDP client and a UDP server in nowait operations.

Table 4 UDP—Nowait Client and Server Steps

Client	Server
1. Optionally, set NonStop TCP/IP, TCPSAM, or TCP6SAM process name (`socket_set_inet_name`).	1. Optionally, set NonStop TCP/IP, TCPSAM, or TCP6SAM process name (`socket_set_inet_name`).
2. a. Create a new socket (`socket_nw`)with (`flags & 0200`) nowait set.b. Call `AWAITIOX`, followed by `SETMODE 30`, followed by `AWAITIOX`).	2. a. Create a new socket (`socket_nw`) with (`flags & 0200`) nowait set.b. Call `AWAITIOX`, followed by `SETMODE 30`, followed by `AWAITIOX`).
3. Optionally, bind the socket to any port (`bind_nw`, followed by `AWAITIOX`).	3. Bind the socket to a well-known port (`bind_nw`, followed by `AWAITIOX`).
4. Start data transfer (`t_sendto_nw` and/or `t_recvfrom_nw`, followed by `AWAITIOX`, usually in a loop). OR Specify the remote address for the socket (`connect_nw`, followed by `AWAITIOX`). Then, start data transfer (`send_nw` and/or `recv_nw`, followed by `AWAITIOX`, usually in a loop).	4. Start data transfer on the new socket (`t_recvfrom_nw` and/or `t_sendto_nw`, followed by `AWAITIOX`, usually in a loop). OR Specify the remote address for the socket (`connect_nw`, followed by `AWAITIOX`). Then, start data transfer on the socket (`recv_nw` and/or `send_nw`, followed by `AWAITIOX`, usually in a loop).
5. Close the socket (`CLOSE or FILE_CLOSE_`).	5. Close the socket (`CLOSE or FILE_CLOSE_`).

Programmatic Interface to Raw Sockets

A raw socket allows direct access to a lower-level protocol—in this case, IP. Access to link-level (Layer 2) protocols is not supported for NonStop TCP/IP, Parallel Library TCP/IP, or NonStop TCP/IPv6. Raw sockets are intended for processes that require the use of some protocol feature not directly accessible through the normal interface, or are intended for the development of new protocols.

Only limited support exists for programming to the raw sockets interface for NonStop TCP/IPv6 and Parallel Library TCP/IP. An application can transmit from any processor using the raw-socket interface but can only receive transmissions in the processor that contains the master TCP6MON or master TCPMON.

Programming at the IP level and using raw sockets requires more work on the part of application clients and servers than programming at the TCP level. First, the application must provide underlying support for whatever transport protocol is used above IP. (For a list of possible protocols, refer to RFC 1010, “Assigned Numbers.”) Then, when performing the basic steps outlined at the beginning of this section, clients and servers must build the transport-level message headers before sending messages, and interpret transport-level message headers and IP headers (including checksums) after receiving the messages. The format for these headers depends on the protocol; for details about the protocol requirements, refer to the appropriate RFC for that protocol.

If your application program refers to a transport protocol by name, the protocol number and name must be included in the file `$SYSTEM.ZTCPIP.PROTOCOL`, as described in the *TCP/IPv6 Configuration and Management Manual*.

Table 5 shows the steps performed by a RAW client and a RAW server in waited operations.

Table 5 RAW—Waited Client and Server Steps

Client	Server
1.	Optionally, set NonStop TCP/IP, TCPSAM, or TCP6SAM process name (`socket_set_inet_name`).
2.	Create a raw socket (`socket`) assigning a protocol number. The default protocol number is 255.
3.	Optionally, bind the socket to any local IP address (`bind`).
4.	Optionally, specify the remote address (`connect`).
5.	If sending messages, perform the following, usually in a loop:
	a. Build the header, as specified by protocol, for type of message being sent.
	b. Start data transfer (`sendto` if `connect` was not called; `send` if `connect` was called).
	If receiving messages, perform the following, usually in a loop:
	a. Start data transfer (`recvfrom` if `connect` was not called; `recv` if `connect` was called).
	b. Read and interpret message header and receive IP header preceding your data.
6.	Close the socket (`CLOSE or FILE_CLOSE_`).

Client	Server
1.	Optionally, set NonStop TCP/IP, TCPSAM, or TCP6SAM process name (`socket_set_inet_name`).
2.	Create a raw socket (`socket`) specifying the protocol number.
3.	Bind the socket to a local IP address (`bind`).
4.	Optionally, specify the remote address (`connect`).
5.	If receiving messages, perform the following, usually in a loop:
	a. Start data transfer (`recvfrom` if `connect` was not called; `recv` if `connect` was called).
	b. Read and interpret message header and interpret IP header.
	If sending messages, perform the following, usually in a loop:
	a. Build the header, as specified by protocol, for type of message being sent.
	b. Start data transfer (`sendto` if `connect` was not called; `send` if `connect` was called).
6.	Close the socket (`CLOSE or FILE_CLOSE_`).

Table 6 shows the steps performed by a RAW client and a RAW server in nowait operations.

Table 6 RAW—Nowait Client and Server Steps

Client	Server
1.	Optionally, set NonStop TCP/IP, TCPSAM, or TCP6SAM process name (`socket_set_inet_name`).
2.	a. Create a raw socket (`socket_nw`)with (`flags & 0200`) nowait set.
	b. Call `AWAITIOX`, followed by `SETMODE 30`, followed by `AWAITIOX`, specifying the protocol number.
3.	Optionally, bind the socket to a local IP address (`bind_nw`, followed by `AWAITIOX`).
4.	Optionally, specify the remote address (`connect_nw`, followed by `AWAITIOX`).
5.	If sending messages, perform the following, usually in a loop:

Client	Server
1.	Optionally, set NonStop TCP/IP, TCPSAM, or TCP6SAM process name (`socket_set_inet_name`).
2.	a. Create a raw socket (`socket_nw`)with (`flags & 0200`) nowait set.
	b. Call `AWAITIOX`, followed by `SETMODE 30`, followed by `AWAITIOX`, specifying the protocol number.
3.	Bind the socket to a local IP address (`bind_nw`, followed by `AWAITIOX`).
4.	Optionally, specify the remote address (`connect_nw`, followed by `AWAITIOX`).
5.	If receiving messages, perform the following, usually in a loop:

Table 6 RAW—Nowait Client and Server Steps *(continued)*

Client	Server		
	a. Build the header, as specified by protocol, for type of message being sent.		a. Start data transfer (`t_recvfrom_nw` if `connect` was not called; `recv_nw` if `connect` was called; each followed by `AWAITIOX`).
	b. Start data transfer (`t_sendto_nw` if `connect` was not called; `send_nw` if `connect` was called; each followed by `AWAITIOX`).		b. Read and interpret message header and interpret IP header.
	If receiving messages, perform the following, usually in a loop:		If sending messages, perform the following, usually in a loop:
	a. Start data transfer (`t_recvfrom_nw` if `connect` was not called; `recv_nw` if `connect` was called; each followed by `AWAITIOX`).		a. Build the header, as specified by protocol, for type of message being sent.
	b. Read and interpret message header and IP header.		b. Start data transfer (`t_sendto_nw` if `connect` was not called; `send_nw` if `connect` was called; each followed by `AWAITIOX`).
6.	Close the socket (`CLOSE or FILE_CLOSE_`).	6.	Close the socket (`CLOSE or FILE_CLOSE_`).

Programming Considerations

When programming your applications, you should consider the following naming convention for the processes and for the handling of buffers in data transfers.

Process Names

All NonStop TCP/IP processes, Parallel Library TCP/IP processes (TCPSAMs), and NonStop TCP/IPv6 processes (TCP6SAMs) have a device type of 48 support calls to the `FILE_GETINFO_` procedure. This provision allows applications to scan for all devices of a specified type, thereby finding all appropriate processes in a system.

NOTE: Parallel Library TCP/IP is only available on NonStop S-series servers.

Multiple NonStop TCP/IP Processes and Logical Network Partitioning (LNP) (NonStop TCP/IPv6, H-Series and G06.22 and Later G-Series RVUs Only)

Logical network partitioning (LNP) is a feature in NonStop TCP/IPv6 that allows you to use the transport-service provider as a way to restrict application access to particular network interfaces. In Parallel Library TCP/IP and in NonStop TCP/IPv6 without LNP configured, all applications in the system have access to all the network interfaces.

When LNP is configured, the NonStop TCP/IPv6 subsystem resembles the conventional NonStop TCP/IP subsystem with multiple TCP/IP processes. The actions necessary to support the application in a multiple NonStop TCP/IP-process environment are similar to the actions necessary to support the application in a multiple-LNP environment.

With LNP configured, applications that initiate connections must select the correct TCP6SAM process as their transport-service provider. The destination IP addresses must be reachable through the transport-service provider of that TCP6SAM. That is, the destination IP addresses must be accessible through the LNP of the TCP6SAM.

For more information about LNP and about selecting the correct TCP6SAM process, see the *TCP/IPv6 Configuration and Management Manual*.

Applications doing ACCEPT_NW2 can only see listening sockets in the same LNP.

Multicasting Operations

Internet Protocol (IP) multicasting provides applications with IP layer access to the multicast capability of Ethernet and networks. IP multicasting, which delivers datagrams on a best-effort basis, avoids the overhead imposed by IP broadcasting on uninterested hosts; it also avoids consumption of network bandwidth by applications that would otherwise transmit separate packets containing identical data to reach several destinations.

IPv4 multicasting achieves efficient multipoint delivery through use of multicast groups. A multicast group is a group of zero or more nodes that is identified by a single Class D IP destination address (IPv4) or a single multicast address (IPv6). An IPv4 Class D address has 1110 in the four high-order bits. In dotted decimal notation, IP multicast addresses range from 224.0.0.0 to 239.255.255.255, with 224.0.0.0 being reserved. An IPv6 multicast address has the format prefix of FF00::/8.

A member of a particular multicast group receives a copy of all data sent to the IP address representing that multicast group. Multicast groups can be permanent or transient. A permanent group has a well-known, administratively assigned IP address. In permanent multicast groups, it is the address of the group that is permanent, not its membership. The number of group members can fluctuate, even dropping to zero.

In IPv4, the All Hosts group (224.0.0.1) and in IPv6 the All Nodes group (FF01::1 (node-local, or scope 1) and FF02::1 (link-local, or scope 2)) multicast addresses are examples of permanent groups. See RFC 1884: IPv6 Addressing Architecture for more information about IPv6 multicast addresses.

IP addresses that are not reserved for permanent multicast groups are available for dynamic assignment to transient groups. Transient groups exist only as long as they have one or more members.

IP multicasting is not supported over connection-oriented transports such as TCP.

NOTE: IP multicasting is implemented using options to the `setsockopt` library call, described in Chapter 4 (page 81). Definitions required for multicast-related socket options are in the <in.h> and <in6.h> header files. Your application must include this header file if you intend that the application receive IP multicast datagrams.

Sending IPv4 Multicast Datagrams

This subsection describe IPv4 only. For information about multicast for IPv6, see Multicast Changes for IPv6 (page 59).

To send IPv4 multicast datagrams, an application indicates the host group to send to by specifying an IP destination address in the range of 224.0.0.0 to 239.255.255.255 in a `sendto` library call. The system maps the specified IP destination address to the appropriate Ethernet multicast address prior to transmitting the datagram.

An application can explicitly control multicast options by using arguments to `setsockopt` library calls. The following options can be set by an application using `setsockopt` library calls:

- Time-to-live field (`IP_MULTICAST_TTL`)
- Multicast interface (`IP_MULTICAST_IF`)
- Disabling loopback of local delivery (`IP_MULTICAST_LOOP`)

NOTE: The syntax for and arguments to the `setsockopt` library call are described in Chapter 4 (page 81). The examples here illustrate how to use the `setsockopt` options that apply to IPv4 multicast datagrams only.

The `IP_MULTICAST_TTL` option to the `setsockopt` library call allows an application to specify a value between 0 and 255 for the time-to-live (TTL) field. Multicast datagrams that have a TTL value of 0 restrict distribution of the multicast datagram to applications running on the local host. Multicast datagrams that have a TTL value of 1 are forwarded only to hosts on the local subnet. If

a multicast datagram has a TTL value greater than 1 and a multicast router is attached to the sending host's network, multicast datagrams can be forwarded beyond the local subnet. Multicast routers forward the datagram to known networks that have hosts belonging to the specified multicast group. The TTL value is decremented by each multicast router in the path. When the TTL value is decremented to 0, the datagram is not forwarded further.

The following example shows how to use the `IP_MULTICAST_TTL` option to the `setsockopt` library call:

```
u_char ttl;
ttl=2;

if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl,
                                    sizeof(ttl)) == -1)
       perror("setsockopt");
```

A datagram addressed to an IP multicast destination is transmitted from the default network interface unless the application specifies that an alternate network interface is associated with the socket. The default interface is determined by the interface associated with the default route in the kernel routing table or by the interface associated with an explicit route, if one exists. Using the `IP_MULTICAST_IF` option to the `setsockopt` library call, an application can specify a network interface other than that specified by the route in the kernel routing table.

The following example shows how to use the `IP_MULTICAST_IF` option to the `setsockopt` library call to specify an interface other than the default:

```
int sock;
struct in_addr ifaddress;
char *if_to_use = "16.141.64.251";

.
.
.
ifaddress.s_addr = inet_addr(if_to_use);
  if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF,
ifaddress,
                                sizeof(ifaddress)) == -1)
     perror ("error from setsockopt IP_MULTICAST_IF");
  else
     printf ("new interface set for sending multicast
              datagrams\n");
```

If a multicast datagram is sent to a group of which the sending host is a member, a copy of the datagram is, by default, looped back by the IP layer for local delivery. The `IP_MULTICAST_LOOP` option to the `setsockopt` library call allows an application to disable this loopback delivery.

The following example shows how to use the `IP_MULTICAST_LOOP` option to the `setsockopt` library call:

```
u_char loop=0;
if (setsockopt( sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop
         sizeof(loop
)) == -1)
 perror("setsockopt");
```

When the value of loop is 0, loopback is disabled. When the value of loop is 1, loopback is enabled. For performance reasons, you should disable the default, unless applications on the same host must receive copies of the datagrams.

Receiving IPv4 Multicast Datagrams

This subsection describe IPv4 only. For information about multicast for IPv6, see Multicast Changes for IPv6 (page 59).

Before a host can receive IP multicast datagrams destined for a particular multicast group, an application must direct the host to become a member of that multicast group. This section describes how an application can direct a host to add itself to and remove itself from a multicast group.

An application can direct the host it is running on to join a multicast group by using the IP_ADD_MEMBERSHIP option to the setsockopt library call as follows:

```
struct ip_mreq mreq;
if (setsockopt( sock, IPPROTO_IP, IP_ADD_MULTICAST, &mreq
         sizeof(mreq
)) == -1)
         perror("setsockopt");
The mreq variable has the following structure:
structip_mreq{
        struct in_addr imr_multiaddr; /* IP multicast
address of group */
        struct in_addr imr_interface; /* local IP
address of interface */
             };
```

Each multicast group membership is associated with a particular interface. The same group can be joined on multiple interfaces. The *imr_interface* variable can be specified as INADDR_ANY, which allows an application to choose the default multicast interface. Alternatively, specifying one of the host's local addresses allows an application to select a particular, multicast-capable interface. The maximum number of memberships that can be added on a single socket is subject to the IP_MAX_MEMBERSHIPS value, which is defined in the <in.h> header file.

To drop membership in a particular multicast group, use the IP_DROP_MEMBERSHIP option to the setsockopt library call:

```
struct ip_mreq mreq;
if (setsockopt( sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq
         sizeof(mreq
))== -1)
 perror("setsockopt");
```

The *mreq* variable contains the same structure values as those values used for adding membership.

If multiple sockets request that a host join a particular multicast group, the host remains a member of that multicast group until the last of those sockets is closed or memberships are dropped from all the sockets.

To receive multicast datagrams sent to a specific UDP port, the receiving socket must have bound to that port using the bind library call. More than one process can receive UDP datagrams destined for the same port if the bind library call (described in Chapter 4) is preceded by a setsockopt library call that specifies the SO_REUSEPORT option. The following example illustrates how to use the SO_REUSEPORT option to the setsockopt library call:

```
int setreuse = 1;
if (setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &setreuse,
                                 sizeof(setreuse)) == -1)
         perror("setsockopt");
```

When the SO_REUSEPORT option is set, every incoming multicast or broadcast UDP datagram destined for the shared port is delivered to all sockets bound to that port.

Delivery of IP multicast datagrams to SOCK_RAW sockets is determined by the protocol type of the destination.

Datagram Protocols and Flow Control

When using datagram protocols, the programmer must manage flow control. Lack of flow control results in the receiver failing to keep up with the sender's rate of transmission, causing a possible overrun condition.

Flow control can be achieved through:

- Rate-based
- Sliding window
- Explicit pacing
- Over subscription (guarantees that the sender cannot overrun the receiver's capacity. The receiver's capacity is greatly in excess of the sender's capacity).

A common misconception states that UDP is more efficient than TCP. However, that idea is only true when you do not need flow control, data and session-loss detection, and accounting for receiving out-of-sequence data. If you do need these properties, you have to provide them programmatically.

However, all flow control must account for the possibility that a datagram could be lost by the network due to congestion or other causes.

UDP is a datagram protocol and TCP is a stream-oriented protocol. TCP is also called connection-oriented while UDP is called connectionless.

TCP guarantees all the properties not supplied by datagram protocols:

- Loss of data detection (delivery assurance)
- Receiving data out of sequence
- Flow control
- Session loss detection
- Congestion avoidance

If these properties are implemented by a higher-level protocol that rides over UDP, you can use UDP. Or, if these properties are not important (as is often the case with broadcast messages) you can use UDP.

Optimal Ways to Deal With Connection Management

Since Guardian does not use signals (like OSS), for Guardian socket programs, the loss of connection may be detected, but is not reportable until the next socket operation so issuing any call might result in an immediate error. So it is possible that on issuing any of the calls, you may get an immediate return indicating an error.

For both OSS and Guardian sockets, if you have lost a connection, send operations may not have made it to the other side before the loss of connection. Therefore, if your application needs to ensure data reception by the other side, you must have a higher-level protocol that has some form of feedback from the other side reflecting positive receipt of the data or the ability to reestablish a synchronization point after the detection of loss of connection. Such a protocol would need, at minimum, sequencing on the data and the ability, when the connection is reestablished, for the receiving side to tell the sending side that it received data up to a specific point or to start over again at a specific point. That process is the reestablishment of synchronization. The higher-level protocol must reestablish synchronization because even TCP does not.

For example, if you are trying to send records of a files to the other side and you send records 1 through 1,000, you could get send completions for everything up to 1,000. But that only means that your TCP/IP stack buffered everything, not that it successfully sent everything. In fact, an error might occur, including loss of connection, after the data has been buffered. So, records 997 through 1,000 would still be sitting in the buffer and you would have no way to know that they never were sent. A higher-level protocol would have numbered the records, then when the loss of connection occurred, it would re-contact the other side and ask which records were sent.

FTP is an example of a higher-level protocol, but it does not do all of these functions. FTP makes you start over from the beginning. FTP establishes synchronization at the end of file. When receiving data, it looks for the start of the file, then everything in between, and then the end of the file. If a

disconnect occurs before the end of file, FTP throws all the data away. FTP is still useful because in fact, loss of connection does not happen often and the cost of retransmission is not always too high. However, a transaction is being transmitted, you must know if it got there and was processed. HP NonStop higher-level protocols frequently used for transaction processing include ODBC and NonStop CORBA which are request/reply model protocols.

Using LISTNER for Custom Applications

If your application fits the standard listener model (see the *TCP/IPv6 Configuration and Management Manual*), you can use LISTNER to start your application programs just like it starts FTPSERV.

Input/Output Multiplexing

Multiplexing is a facility used in applications to transmit and receive I/O requests among multiple sockets. HP NonStop systems support this facility with nowaited operations which also allow you to multiplex socket I/O with other kinds of I/O. The new IPv6 library routines have not been implemented in nowaited form. See Optimal Ways to Deal With Connection Management (page 47) for information about nowaited operations.)

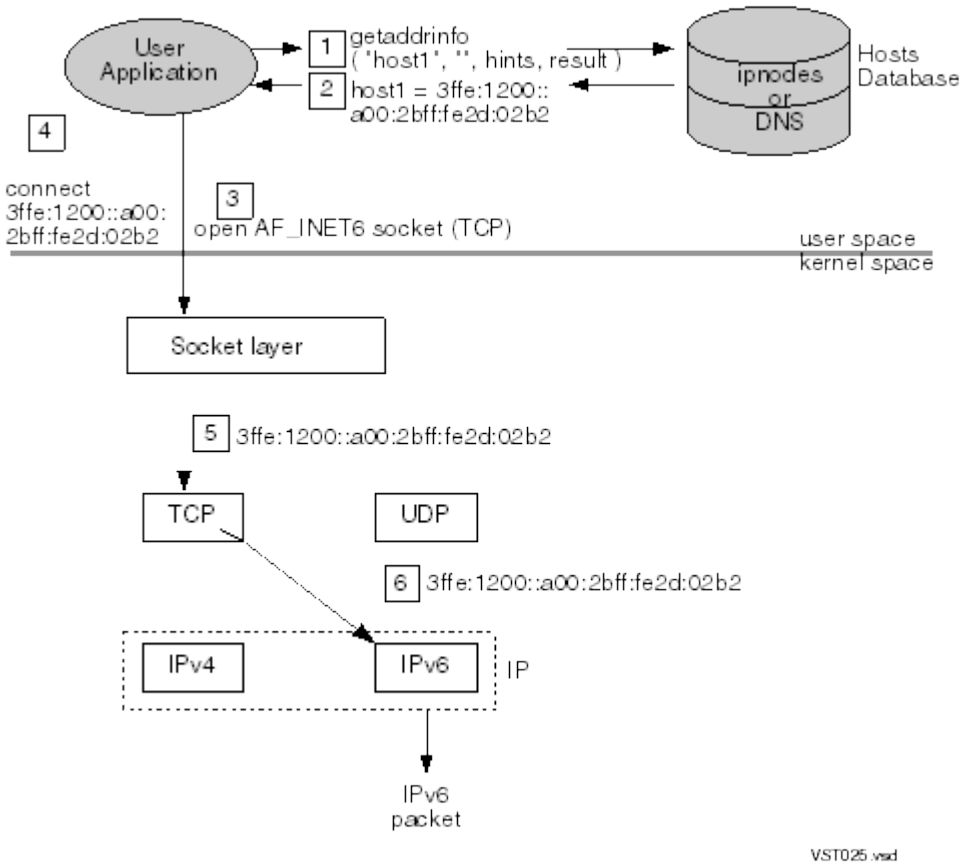
2 Porting and Developing IPv6 Applications (NonStop TCP/IPv6 and CIP Only)

This section explains how to write Guardian socket applications for IPv4 and IPv6 communications. Topics include:

Using AF_INET6-Type Guardian Sockets for IPv6 Communications

You can use AF_INET6-type Guardian sockets for IPv6 communication as well as for IPv4 communication. Table 7 (page 53) shows the sequence of events for a client application that uses an AF_INET6-type Guardian socket to send IPv6 packets.

Figure 2 Using AF_INET6 Sockets for IPv6 Communications



1. Application calls getaddrinfo and passes the hostname (host1), the AF_INET6 address family hint, and the AI_ADDRCONFIG flag hints. The flag hints tell the function that if an IPv6 address is found for host1, return it. See addrinfo for a description of hints fields and values.
2. The search finds an IPv6 address for host1 in the hosts database, and getaddrinfo returns the IPv6 address 3ffe:1200::a00:2bff:fe2d:02b2 in one or more structures of type addrinfo.

3. The application calls socket to create an AF_INET6 socket, using the address family and socket type contained in the addrinfo structure.
4. If the socket call is successful, the application calls connect to establish a connection with host1, using the host address and length in the addrinfo structure. If the connect call is successful, the application sends information to the 3ffe:1200::a00:2bff:fe2d:02b2 address.

   NOTE: After using the information in the addrinfo structures, the application calls freeaddrinfo to free system resources used by the structures.

5. The socket layer passes the information and address to the UDP module.
6. The UDP module identifies the IPv6 address, puts the 3ffe:1200::a00:2bff:fe2d:02b2 address into the packet header, and passes the information to the IPv6 module for transmission.

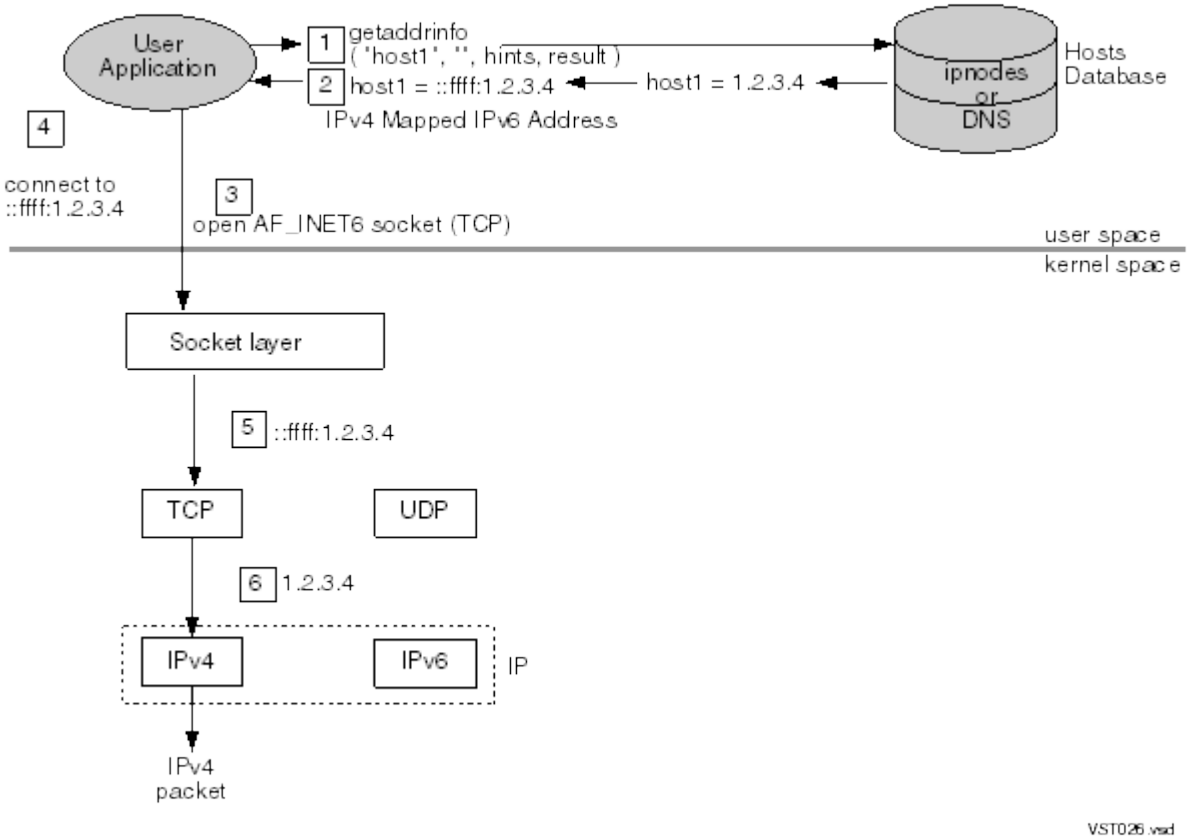
From this point, the application can do the following:

- Call recv to wait for a response from the server system.
- After the application receives a response, call getpeername, getpeername_nw to determine the address of the connected socket. The address is returned in a structure of type sockaddr_in6.
- Call getnameinfo using the NI_NAMEREQD flag to obtain the server name.
- Call getnameinfo using the NI_NUMERICHOST flag to convert the server address to a text string. Chapter 5 contains sample client program code that demonstrates these steps.

Using AF_INET6 Guardian Sockets for IPv4 Communications

You can also use an AF_INET6 socket for IPv4 communications. Figure 3 (page 50) shows the sequence of events for a client application that uses an AF_INET6 socket to send IPv4 packets. (For information about IPv4 mapped IPv6 addresses, see the *TCP/IPv6 Configuration and Management Manual*.)

Figure 3 Using AF_INET6 Sockets for IPv4 Communications (Send)



1. The application calls getaddrinfo and passes the hostname (host1), the AF_INET6 address family hint, and the AI_ADDRCONFIG and AI_V4MAPPED flag hints. The flag hints tell the function that if an IPv4 address is found for host1, return it as an IPv4-mapped IPv6 address. See addrinfo for a description of hints fields and values.
2. The search finds an IPv4 address, 1.2.3.4, for host1 in the hosts database, and getaddrinfo returns the IPv4-mapped IPv6 address ::ffff:1.2.3.4 in one or more structures of type addrinfo.
3. The application calls socket to create an AF_INET6 socket, using the address family and socket type contained in the addrinfo structure. The socket is a datagram socket (UDP) in this example, but could be a stream socket (TCP).
4. If the socket call is successful, the application calls connect to establish a connection to host1, using the host address and length in the addrinfo structure. If the connect call is successful, the application sends information to the ::ffff:1.2.3.4 address.

   NOTE: After using the information in the addrinfo structures, the application calls freeaddrinfo to free system resources used by the structures.

5. The socket layer passes the information and address to the UDP module.
6. The TCP module identifies the IPv4-mapped IPv6 address, puts the 1.2.3.4 address into the packet header, and passes the information to the IPv4 module for transmission.

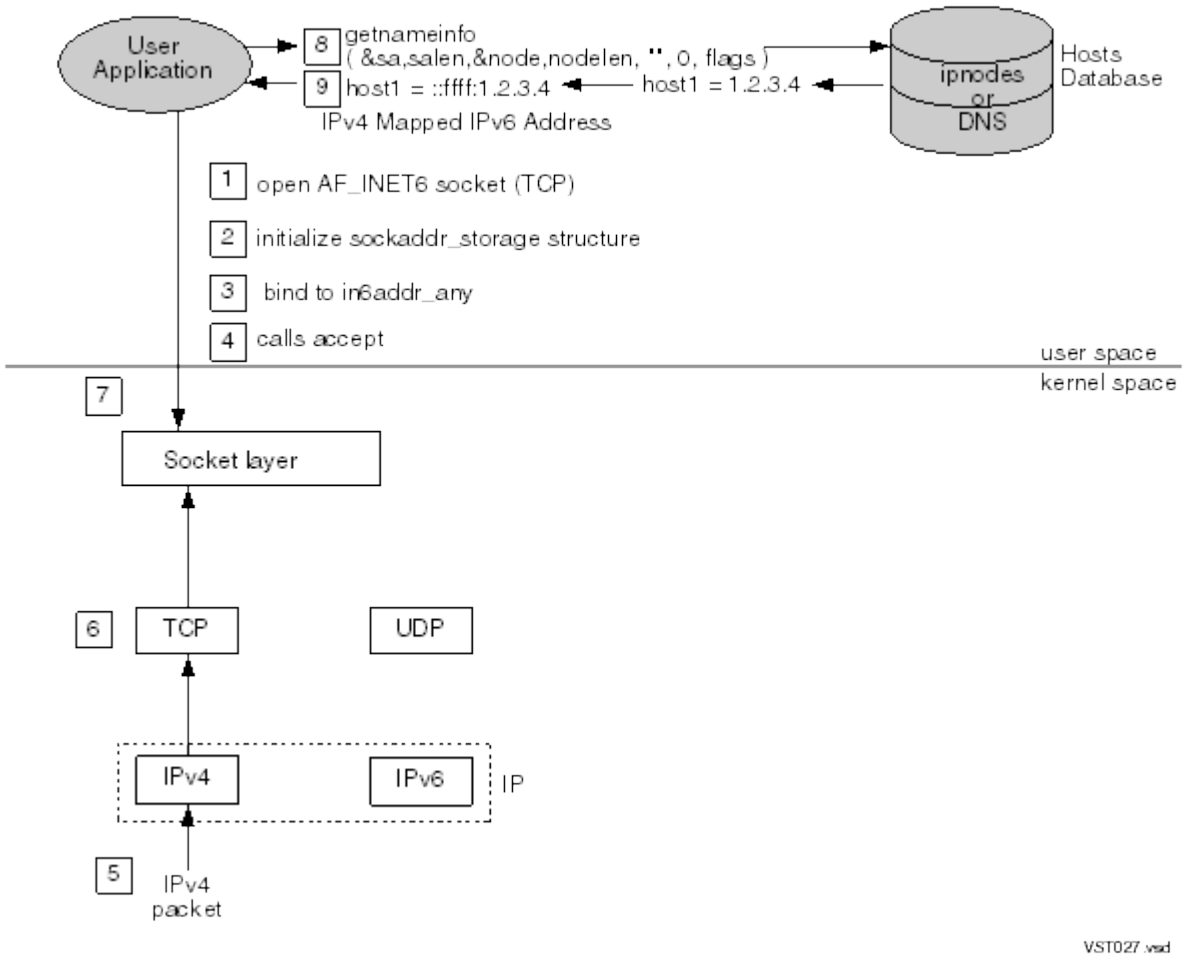
From this point, the application can do the following:

- Call recv to wait for a response from the server system.
- After the application receives a response, call getpeername to determine the address of the connected socket. The address is returned in a structure of type sockaddr_in6. If you want your application to be protocol-independent, use the sockaddr_storage structure instead of the sockaddr_in6 structure.
- Call getnameinfo using the NI_NAMEREQD flag to obtain the server name.
- Call getnameinfo using the NI_NUMERICHOST flag to convert the server address to a text string. Chapter 5 contains sample client program code that demonstrates these steps.

Using AF_INET6 Guardian Sockets to Receive Messages

AF_INET6 sockets can receive messages sent to either IPv4 or IPv6 addresses. An AF_INET6 socket uses the IPv4-mapped IPv6 address format to represent IPv4 addresses. Figure 2-3 shows the sequence of events for a server application that uses an AF_INET6 socket to receive IPv4 packets.

Figure 4 Using AF_INET6 Socket for IPv4 Communications (Receive)



1. The application calls socket to create an AF_INET6 socket.
2. The application initializes a sockaddr_storage structure, and sets the family, address, and port.
3. The application calls bind to assign in6addr_any to the socket.
4. The application calls accept to mark the socket to listen and wait for incoming connections.
5. An IPv4 packet arrives and passes through the IPv4 module.
6. The TCP layer strips off the packet header and passes the information and the IPv4-mapped address (::ffff:1.2.3.4) to the socket layer.
7. The socket layer returns the information to the application. The information from the socket is passed to the application in a sockaddr_storage structure. (Using sockaddr_storage instead of sockaddr_in6 makes the application protocol-independent.)
8. The application calls getnameinfo and passes the ::ffff:1.2.3.4 address and the NI_NAMEREQD flag. The flag tells the function to return the hostname for the address. See getnameinfo (page 117) for a description of the flags bits and their meanings.
9. The search finds the hostname for the 1.2.3.4 address in the hosts database, and getnameinfo returns the hostname.

Chapter 5 contains sample server program code that demonstrates these steps.

Address-Testing Macros

In some cases, an application that uses an AF_INET6 socket for communications needs to determine the type of address that is returned in the structure. For this case, the API defines macros to test the

addresses. Table 7 lists the currently defined address-testing macros and the return value for a valid test. To use these macros, include the following file in your application:

```
#include <in6.h>
```

The address-testing macros return true if the address is of the specified type, otherwise, they return false. The scope-testing macros test the scope of a multicast address and return true if the address is a multicast address of the specified scope or false if the address is either not a multicast address or not of the specified scope. IN6_IS_ADDR_LINKLOCAL and IN6_IS_ADDR_SITELOCAL return true only for the two local-use IPv6 unicast addresses; these two macros do not return true for IPv6 multicast addresses of either link-local scope or site-local scope.

Table 7 Address and Scope-Testing Macros

Address-Testing Macros	Scope-Testing Macros
`int IN6_IS_ADDR_UNSPECIFIED (const struct in6_addr *);`	`int IN6_IS_ADDR_MC_NODELOCAL (const struct in6_addr *);`
`int IN6_IS_ADDR_LOOPBACK (const struct in6_addr *);`	`int IN6_IS_ADDR_MC_LINKLOCAL (const struct in6_addr *);`
`int IN6_IS_ADDR_MULTICAST (const struct in6_addr *);`	`int IN6_IS_ADDR_MC_SITELOCAL (const struct in6_addr *);`
`int IN6_IS_ADDR_LINKLOCAL (const struct in6_addr *);`	`int IN6_IS_ADDR_MC_ORGLOCAL (const struct in6_addr *);`
`int IN6_IS_ADDR_SITELOCAL (const struct in6_addr *);`	`int IN6_IS_ADDR_MC_GLOBAL (const struct in6_addr *);`
`int IN6_IS_ADDR_V4MAPPED (const struct in6_addr *);`	
`int IN6_IS_ADDR_V4COMPAT (const struct in6_addr *);`	

Porting Applications to Use AF_INET6 Sockets

AF_INET6 sockets enable applications to communicate using the IPv6 protocol, IPv4 protocol, or both. For IPv6 communication, RFC 2553, Basic Socket Interface Extensions for IPv6, specifies changes to the BSD socket Applications Programming Interface (API). Table 2-2 summarizes these changes.

Table 8 Summary of IPv6 Extensions to the BSD Socket API

Category	Changes
Core function calls	None; basic syntax of socket functions stays the same. Applications must cast pointers to the protocol-specific address structures into pointers to the generic sockaddr address structure when using the socket functions. See Making Structure Changes (page 54) for information on creating Internet applications.
Socket address structure	Specifies a new `sockaddr_in6` structure for IPv6 communications and a `sockaddr_storage` structure for protocol-independent communication. The `sockaddr_in` structure remains for IPv4 communications. See Making Structure Changes (page 54) for more information.
Name-to-address translation	Specifies the `getnameinfo`, `getaddrinfo`, `getipnodebyname`, and `getipnodebyaddr` functions for protocol-independent (IPv4 and IPv6) communication. The `gethostbyaddr` and `gethostbyname` functions remain for IPv4 communications only. See Making Library Routine Changes (page 56) for more information.
Address conversion functions	Specifies the `inet_pton` and `inet_ntop` functions for protocol-independent (IPv4 and IPv6) address conversion. The `inet_ntoa`

Table 8 Summary of IPv6 Extensions to the BSD Socket API *(continued)*

Category	Changes
	and `inet_addr` functions remain for IPv4 address conversion only. See Making Library Routine Changes (page 56) for more information.
Socket options	Specifies new socket options for IPv6 multicast. See Multicast Changes for IPv6 (page 59) for more information.

Application Changes

This subsection describes the changes you must make in your existing application code in order to operate in an IPv6 networking environment. When you have finished porting your applications to IPv6, any existing IPv4 applications continue to operate as before and also interoperate with your IPv6 application.

Changes to your applications described in this subsection include:

- Making Name Changes
- Making Structure Changes
- Making Library Routine Changes (page 56)
- Making Other Application Changes (page 57)

Making Name Changes

Most changes required are straightforward and mechanical but some may require some code restructuring. (For example, a routine that returns an int datatype holding an IPv4 address may need to be modified to take a pointer to an in6_addr structure as an extra parameter into which it writes the IPv6 address). Table 9 summarizes the changes to make to your application's code.

Table 9 Name Changes

Search file for:	Replace with:
AF_INET	AF_INET6
PF_INET	PF_INET6
INADDR_ANY	in6addr_any

Making Structure Changes

The structure names and field names have changed for the following structures:

- `in_addr`
- `sockaddr_in`
- `sockaddr`
- `hostent`

in_addr Structure Changes for Protocol-Independent Applications

Applications that use the in_addr structure must be changed, as needed, to use the in6_addr structure, as shown in the following examples:

AF_INET Structure	AF_INET6 Structure
struct in_addr	struct in6_addr
unsigned long s_addr	u_char sa6_addr

Make the following changes in your application, as needed:

	Original	Change to
Structure Name	in_addr	in6_addr
Data Type	unsigned long	u_char
Field Name	s_addr	sa6_addr

See Making Other Application Changes (page 57) for additional changes you might need to make to your application. See also in6_addr (page 70) for alternative definitions of the in6_addr data structure.

sockaddr_in Structure Changes for IPv6 Applications

Applications that use the 4.4 BSD `sockaddr_in` structure must be changed, as needed, to use the `sockaddr_in6` structure for IPv6 sockets as shown in the following examples:

AF_INET Structure	AF_INET6 Structure	Comment
struct sockaddr_in unsigned char sin_len sa_family_t sin_family in_port_t sin_port struct in_addr sin_addr	struct sockaddr_in6 uint8_t sin6_len sa_family_t sin6_family int_port_t sin6_port struct in6_addr sin6_addr	length of this struct (24)AF_INET6 familytransport layer port #IPv6 address

NOTE: In addition to the fields shown above for INET6, there are two new fields in INET6: sin6_flowinfo and sin6_scope_id. See sockaddr_in6 (page 78).

Make the following change in your application, as needed:

	Original	Change to
Structure Name	`sockaddr_in`	`sockaddr_in6`
Data Type/Field Name	`unsigned char sin_len`	`u_int8_t sin6_len`
Data Type/Field Name	`sa_family_t sin_family`	`sa_family_t sin_family`
Data Type/Field Name	`in_port_t sin_port`	`int_port_t sin6_port`
Data Type/Field Name	`struct in_addr sin_addr`	`struct in6_addr sin6_addr`

Applications that use the 4.3 BSD `sockaddr_in` structure must be changed, as needed, to use the `sockaddr_in6` structure for IPv6 sockets as shown in the following examples:

AF_INET Structure	AF_INET6 Structure
struct sockaddr_in u_short sin_family in_port_t sin_port struct in_addr sin_addr	struct sockaddr_in6 u_short sin6_family in_port_t sin6_port struct in6_addr sin6_addr

NOTE: In addition to the fields shown above for INET6, there are two new fields in INET6: sin6_flowinfo and sin6_scope_id. See sockaddr_in6 (page 78).

Make the following change in your application, as needed:

	Original	Change to
Structure Name	`sockaddr_in`	`sockaddr_in6`
Data Type/Field Name	`u_short sin_family`	`u_short sin6_family`
Data Type/Field Name	`in_port_t sin_port`	`in_port_t sin6_port`
Data Type/Field Name	`struct in_addr sin_addr`	`struct in6_addr sin6_addr`

NOTE: In both cases, you should initialize the entire `sockaddr_in6` structure to zero after your structure declarations.

Making Library Routine Changes

You must make changes, as needed, to applications that use the following library routines:

- `gethostbyaddr`
- `gethostbyname`
- `inet_ntoa`
- `inet_addr`

gethostbyaddr Function Call

Change applications that use the `gethostbyaddr` function call to use the `getnameinfo` function call, as shown in the following examples:

AF_INET Call	AF_INET6 Call
gethostbyaddr(xxx,4,AF_INET)	getnameinfo(*&sockaddr, sockaddr_len, node_name, name_len,* *service, service_len, flags*);

Make the following changes in your application, as needed:

Change the function name from `gethostbyaddr` to `getnameinfo` and provide a pointer to the socket address structure, a character string for the returned node name, an integer for the length of the returned node name, a character string to receive the returned service name, an integer for the length of the returned service name, and an integer that specifies the type of address processing to be performed.

Alternatively, you can use `getipnodebyaddr`. The difference between `getnameinfo` and `getipnodebyaddr` is that `getnameinfo` returns both the node name and the services name and `getipnodebyaddr` returns just the node name. `getipnodebyaddr` also requires another call, `freehostent`, to free the `hostent` structure when the call is complete.

See getnameinfo (page 117) and getipnodebyaddr (page 114) for more information about these library routines.

gethostbyname Function Call

Applications that use the `gethostbyname` function call must be changed to use the `getaddrinfo` function call, as shown in the following examples:

AF_INET Call	AF_INET6 Call

gethostbyname(name)	getaddrinfo(*node_name, service_name, &hints, &result*);
	.
	.
	.
	freeaddrinfo(result);

Make the following changes in your application, as needed:

1. Change the function name from `gethostbyname` to `getaddrinfo`.
2. Provide:
   - a character string for the returned node name
   - a character string for the service name
   - a pointer to a hints structure that contains processing options
   - a pointer to an `addrinfo` structure or structures for the returned address information. (See getaddrinfo (page 107) for a description of hints fields and values.)
3. Add a call to the `freeaddrinfo` routine to free the `addrinfo` structure or structures when your application is finished using them.

Alternatively, you can use `getipnodebyname`. The difference between `getaddrinfo` and `getipnodebyname` is that `getaddrinfo` returns both the node address and the port number and `getipnodebyaddr` returns just the node address. `getipnodebyname` also requires another call, `freehostent`, to free the `hostent` structure when the call is complete.

See getaddrinfo and getipnodebyname for more information about these calls.

If your application needs to determine whether an address is an IPv4 address or an IPv6 address, and cannot determine this information from the address family, use the `IN6_IS_ADDR_V4MAPPED` macro. See Address-Testing Macros (page 52) for a list of IPv6 address testing macros.

inet_ntoa Function Call

Applications that use the `inet_ntoa` function call must be changed to use the `inet_ntop` function call, as shown in the following examples:

AF_INET Call	AF_INET6 Call
inet_ntoa(addr)	inet_ntop(family, &addr, &buff, len)

In your applications, change the function name from `inet_ntoa` to `inet_ntop` and provide the family name (AF_INET or AF_INET6), the address of the input buffer containing the binary address, a non-NULL address, and the size of the address to convert. See inet_ntop (page 138) for a description of the library routine.

inet_addr Function Call

Applications that use the `inet_addr` function call must be changed to use the `inet_pton` function call, as shown in the following examples:

AF_INET Call	AF_INET6 Call
result=inet_addr(&string);	result=inet_pton(family, &addr, &buff)

Make the following changes in your application, as needed:

Change the function name from `inet_addr` to `inet_pton` and provide the family name (AF_INET or AF_INET6), the address of the address string containing to be converted, and the address of the buffer into which the function stores the numeric address upon return. See inet_pton (page 139) for a description of hints fields and values.

Making Other Application Changes

In addition to the name changes, you should review your code for specific uses of IP address information and variables.

Comparing IP Addresses

If your application compares IP addresses or tests IP addresses for equality, the `in6_addr` structure changes you made in Making Structure Changes (page 54) change the comparison of `int` quantities to a comparison of structures. This modification breaks the code and causes compiler errors.

Make either of the following changes to your application, as needed:

AF_INET Code	AF_INET6 Code
(addr1->s_addr == addr2->s_addr)	(memcmp(addr1, addr2, sizeof(struct in6_addr)) == 0)

Change the equality expression to one that uses the `memcmp` (memory comparison) function.

AF_INET Code	AF_INET6 Code
(addr1->s_addr == addr2->s_addr)	IN6_ARE_ADDR_EQUAL(addr1, addr2)

Change the equality expression to one that uses the `IN6_ARE_ADDR_EQUAL` macro. See Address-Testing Macros (page 52) for a list of IPv6 address testing macros.

Comparing an IP Address to the Wild Card Address

If your application compares an IP address to the wild card address, the `in6_addr` structure changes you made in Making Structure Changes (page 54) change the comparison of `int` quantities to a comparison of structures. This modification breaks the code and cause compiler errors.

Make either of the following changes to your application, as needed:

AF_INET Code	AF_INET6 Code
(addr->s_addr == INADDR_ANY)	IN6_IS_ADDR_UNSPECIFIED(addr)

Change the equality expression to one that uses the `IN6_IS_ADDR_UNSPECIFIED` macro. See Address-Testing Macros (page 52) for a list of IPv6 address testing macros.

AF_INET Code	AF_INET6 Code
(addr->s_addr == INADDR_ANY)	(memcmp(addr, in6addr_any, sizeof(struct in6_addr)) == 0)

Change the equality expression to one that uses the memcmp (memory comparison) function.

Using int Data Types to Hold IP Addresses

If your application uses `int` data types to hold IP addresses, the `in6_addr` structure changes you made in Making Structure Changes (page 54) changes the assignment. This modification breaks the code and causes compiler errors.

Make the following changes to your application, as needed:

AF_INET Code	AF_INET6 Code
struct in_addr foo;	struct in6_addr foo
int bar;	struct in6_addr bar;
.	.
.	.
.	.
bar = foo.s_addr;	bar = foo;

1. Change the data type for `bar` from `int` to a `struct in6_addr`.
2. Change the assignment statement for `bar` to remove the `s_addr` field reference.

Using Functions That Return IP Addresses

If your application uses functions that return IP addresses as int data types, the in6_addr structure changes you made in Making Structure Changes (page 54) changes the destination of the return value from an `int` to an array of `char`. This modification breaks the code and causes compiler errors.

Make the following changes to your application, as needed:

AF_INET Code	AF_INET6 Code
struct in_addr *addr;	struct in6_addr *addr;
addr->s_addr = foo(*xxx*);	foo(*xxx*, *addr*);

Restructure the function to enable you to pass the address of the structure in the call. In addition, modify the function to write the return value into the structure pointed to by addr.

Changing Socket Options

If your application uses IPv4 IP-level socket options, change them to the corresponding IPv6 options. See setsockopt, setsockopt_nw (page 184) for more information.

Multicast Changes for IPv6

This subsection describes changes you need to make to perform multicast communications in IPv6. This subsection describe IPv6 sending and receiving only. For information about multicast for IPv4 as well as overview information about IPv6 multicast communications, see Multicasting Operations (page 44).

Sending IPv6 Multicast Datagrams

To send IPv6 multicast datagrams, an application indicates the multicast group to send to by specifying an IPv6 multicast address in a `sendto` library call. (See sendto (page 177).) The system maps the specified IPv6 destination address to the appropriate Ethernet or FDDI multicast address prior to transmitting the datagram.

An application can explicitly control multicast options by using arguments to set the following options in the `setsockopt` and `setsockopt_nw` library calls:

- Hop limit (`IPV6_MULTICAST_HOPS`)
- Multicast interface (`IPV6_MULTICAST_IF`)
- Disabling loopback of local delivery (`IPV6_MULTICAST_LOOP`)

NOTE: The syntax for and arguments to the `setsockopt` library call are described in setsockopt, setsockopt_nw (page 184). The examples here and in Chapter 4 illustrate how to use the `setsockopt` options that apply to IPv6 multicast datagrams only.

The `IPV6_MULTICAST_HOPS` option to the setsockopt library call allows an application to specify a value between 0 and 255 for the hop limit field.

- Multicast datagrams that have a hop limit value of 0 restrict distribution of the multicast datagram to applications running on the local host.
- Multicast datagrams that have a hop limit value of 1 are forwarded only to hosts on the local link.

If a multicast datagram has a hop limit value greater than 1 and a multicast router is attached to the sending host's network, multicast datagrams can be forwarded beyond the local link. Multicast routers forward the datagram to known networks that have hosts belonging to the specified multicast group.

The hop limit value is decremented by each multicast router in the path. When the hop limit value is decremented to 0, the datagram is not forwarded further.

The following example shows how to use the `IPV6_MULTICAST_HOPS` option to the `setsockopt` library call:

```
setsockopt library call:
u_char hops;
hops=2;
if (setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_HOPS, &hops,
                                    sizeof(hops)) < 0)
       perror("setsockopt: IPV6_MULTICAST_HOPS error");
```

A multicast datagram addressed to an IPv6 multicast address is transmitted from the default network interface unless the application specifies that an alternate network interface is associated with the socket. The default interface is determined by the interface associated with the default route in the kernel routing table or by the interface associated with an explicit route, if one exists. Using the `IPV6_MULTICAST_IF` option to the `setsockopt` library call, an application can specify a network interface other than that specified by the route in the kernel routing table.

The following example shows how to use the `IPV6_MULTICAST_IF` option to the `setsockopt` library call to specify an interface other than the default:

```
u_int if_index = 1;
.
.
.
if (setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_IF, &if_index,
                             sizeof(if_index)) < 0)
   perror ("setsockopt: IPV6_MULTICAST_IF error");
else
   printf ("new interface set for sending multicast datagrams\n");
```

The `if_index` parameter specifies the interface index of the desired interface or 0 to select a default interface. You can use the `if_nametoindex` routine to find the interface index.

If a multicast datagram is sent to a group of which the sending node is a member, a copy of the datagram is, by default, looped back by the IP layer for local delivery. The `IPV6_MULTICAST_LOOP` option to the `setsockopt` library call allows an application to disable this loopback delivery.

The following example shows how to use the `IPV6_MULTICAST_LOOP` option to the `setsockopt` library call:

```
u_char loop=0;
if (setsockopt( sock, IPPROTO_IPV6, IPV6_MULTICAST_LOOP, &loop,
         sizeof(loop)) < 0)
         perror("setsockopt: IPV6_MULTICAST_LOOP error");
```

When the value of *`loop`* is 0, loopback is disabled. When the value of *`loop`* is 1, loopback is enabled. For performance reasons, you should disable the default, unless applications on the same host must receive copies of the datagrams.

Receiving IPv6 Multicast Datagrams

Before a node can receive IPv6 multicast datagrams destined for a particular multicast group other than the `all nodes` group, an application must direct the node to become a member of that multicast group.

This subsection describes how an application can direct a node to add itself to and remove itself from a multicast group.

An application can direct the node it is running on to join a multicast group by using the `IPV6_JOIN_GROUP` option to the `setsockopt` library call as follows:

```
struct ipv6_mreq imr6;
.
.
```

```
.
imr6.ipv6mr_interface = if_index;
if (setsockopt( sock, IPPROTO_IPV6, IPV6_JOIN_GROUP,
     (char *)&imr6, sizeof(imr6)) < 0)
     perror("setsockopt: IPV6_JOIN_GROUP error");
```

The / variable has the following structure:

```
struct ipv6_mreq {
        struct in6_addr ipv6mr_multiaddr; /*IP multicast address of group*/
        unsigned int ipv6mr_interface; /*local interface index*/
        };
```

Each multicast group membership is associated with a particular interface. It is possible to join the same group on multiple interfaces. The *ipv6mr_interface* variable can be specified with a value of 0, which allows an application to choose the default multicast interface. Alternatively, specifying one of the host's local interfaces allows an application to select a particular, multicast-capable interface. The maximum number of memberships that can be added on a single socket is subject to the IPV6_MAX_MEMBERSHIPS value, which is defined in the <in6.h> header file.

Dropping Membership in a Multicast Group

To drop membership in a particular multicast group use the IPV6_LEAVE_GROUP option to the setsockopt library call (see setsockopt, setsockopt_nw (page 184)):

```
struct ipv6_mreq imr6;
if (setsockopt( sock, IPPROTO_IPV6, IPV6_LEAVE_GROUP, &imr6,
     sizeof(imr6)) < 0)
     perror("setsockopt: IPV6_LEAVE_GROUP error");
```

The imr6 parameter contains the same structure values used for adding membership.

If multiple sockets request that a node join a particular multicast group, the node remains a member of that multicast group until the last of those sockets is closed.

To receive multicast datagrams sent to a specific UDP port, the receiving socket must have bound to that port using the bind library call. More than one process can receive UDP datagrams destined for the same port if the bind library call (described in Chapter 4) is preceded by a setsockopt library call that specifies the SO_REUSEPORT option.

Delivery of IP multicast datagrams to SOCK_RAW sockets is determined by the protocol type of the destination.

3 Data Structures

This section describes the library header files and the data structures declared in the headers. The function declarations and data structures contained in the header files are used by the socket library routines described in Chapter 4.

Library Headers

The declarations of the functions in the socket library are provided in both C and TAL programming languages. Other languages can be used to interface to the socket library, subject to C compiler restrictions.

NOTE: Use the Common Run-Time Environment (CRE) when developing TAL socket applications. CRE is described in the *CRE Programming Manual*.

All TAL declarations are in the `$SYSTEM.ZTCPIP.SOCKDEFT` file.

Each C header contains declarations for a related set of library functions, as well as constants and structures that enhance that set. When you use a routine in the socket library, you must first make sure that you have included the headers listed in the `#include` directives that precede the calling syntax for that routine (see the syntax boxes in Chapter 4).

You should not declare the routine itself because the header files contain declarations for the routines. Header declarations include directives stating whether you are compiling for the large-memory model or the wide-data model.

The socket library header files are supplied in the subvolume, `$SYSTEM.ZTCPIP`.

Table 10 lists the NonStop TCP/IP, Parallel Library TCP/IP, and NonStop TCP/IP C header files and their contents. (TCP/IP/PL in this table denotes Parallel Library TCP/IP.)

NOTE: Parallel Library TCP/IP is only available on NonStop S-series servers.

Table 10 Summary of C Header Files and Contents

Header Files	Subsystem	Contents
if.h	TCP/IP, TCP/IP/PL, TCP/IPv6	Structures defining the network-level interface
in.h	TCP/IP, TCP/IP/PL, TCP/IPv6	Constants and structures defined by the Internet system
in6.h	TCP/IPv6	Constants and structures for IPv6.
ioctl.h	TCP/IP, TCP/IP/PL, TCP/IPv6	I/O control definitions
netdb.h	TCP/IP, TCP/IP/PL, TCP/IPv6	Structures returned by the network database library
route.h	TCP/IP, TCP/IP/PL, TCP/IPv6	Definitions related to routing tables
socket.h	TCP/IP, TCP/IP/PL, TCP/IPv6	Definitions related to sockets: types, address families, options

Some of the following C header files are used internally by the NonStop TCP/IP, Parallel Library TCP/IP, and NonStop TCP/IPv6 subsystems; others are useful in some application programs. The files are user-readable and contain comments describing their contents, as follows:

af.h	insystm.h	nameser.h	sockvar.h	tcpseq.h	udpvar.h
domain.h	invar.h	netisr.h	syscal.h	tcptimr.h	uio.h
icmpvar.h	ip.h	param.h	tcp.h	tcpvar.h	user.h
ifarp.h	ipicmp.h	protosw.h	tcpdeb.h	time.h	
ifether.h	ipvar.h	rawcb.h	tcpfsm.h	types.h	
inpcb.h	mbuf.h	resolv.h	tcpip.h	udp.h	

Data Structures

Several important data structures are used by the socket library routines. The data structures are provided in the header files in the $SYSTEM.ZTCPIP subvolume. Table 11 lists the data structures, indicating the page where its documented and the C header file in which each structure is declared as well as the type of routine that uses that structure.

Table 11 Summary Data Structures and C Header Files

Structure	Header File	Type of Routine That Uses Structure
addrinfo (page 64)	netdb.h	Support
arpreq (page 65)	ifarp.h	Socket I/O
hostent (page 66)	netdb.h	Support
if_nameindex (page 67)*	if.h	Socket I/O
ifreq (page 68)	if.h	Socket I/O
in_addr (page 69)	in.h	Socket
in6_addr (page 70)*	in.h	Socket
ipv6_mreq (page 71)*	in.h	Socket I/O
netent (page 71)	netdb.h	Support
open_info_message (page 72)	netdb.h	Support
protoent (page 73)	netdb.h	Support
rtentry (page 74)	route.h	Socket I/O
send_nw_str (page 75)	netdb.h	Socket (send_nw)
sendto_recvfrom_buf (page 76)	in.h	Socket
servent (page 76)	netdb.h	Support
sockaddr (page 77)	netdb.h	Socket
sockaddr_in (page 78)	in.h	Socket
sockaddr_in6 (page 78)*	in.h	Socket
sockaddr_storage (page 79)*	in.h	Socket

* Applies to NonStop TCP/IPv6 only

See Chapter 4 (page 81), for more information about the different types of socket library calls; the socket function calls are listed in Table 12 (page 82). The socket I/O control operations are socket_ioctl and socket_ioctl_nw. The socket I/O control operations and the structures they use are listed in Table 16 (page 199).

The data structures used by the support routines are built from the following data files:

- `$SYSTEM.ZTCPIP.HOSTS`
- `$SYSTEM.ZTCPIP.IPNODES (NonStop TCP/IPv6 only)`
- `$SYSTEM.ZTCPIP.SERVICES`
- `$SYSTEM.ZTCPIP.NETWORKS`
- `$SYSTEM.ZTCPIP.PROTOCOL`

The formats of these four data files are given in the *TCP/IPv6 Configuration and Management Manual*.

In this section, the description of each structure includes the following information:

- Purpose of the structure
- Structure declaration (enclosed in a box), for both C and TAL
- Description of each field in the structure declaration
- Type of routine that uses the structure

The structure descriptions are arranged alphabetically.

addrinfo

The address info structure is used by the network database library. This structure is defined in the `netdb.h` header file. Use this structure in applications that assume some of the functions of a transport protocol such as TCP or UDP.

C Declaration

```
#include <netdb.h>
struct addrinfo {
   int      ai_flags;         /*input flags */
   int      ai_family;        /*protofamily for socket */
   int      ai_socktype;      /* socket type */
   int      ai_protocol;      /* protocol for socket */
   size_t   ai_addrlen;       /* length of socket address /*
   char     *ai_canonname;    /* ptr to canonical name for host*/
   struct sockaddr *ai_addr;  /* ptr to socket address structure */
   struct addrinfo *ai_next;  /* pointer to next in list */
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT addrinfo (*);
 INT(32)      ai_flags;
 INT(32)      ai_family;
 INT(32)      ai_socktype;
 INT(32)      ai_protocol;
 INT(32)      ai_addrlen;
 STRING .EXT ai_canonname;
 INT(32).EXT ai_addr (sockaddr);
 INT(32).EXT ai_next(addrinfo);
```

ai_flags

contains a combination of one or more of the following flags:

AI_PASSIVE	Returns an address that can be passed to the `bind` function. If `hostname` is `NULL`, the address is set to `INADDR_ANY` or `in6addr_any`, as appropriate for the address family. If this flag is

	not set, the returned address can be passed to the connect function. If `hostname` is `NULL`, the address is set to the loopback address.
AI_CANONNAME	Requests the return of the canonical name for the host if the *`hostname`* is not NULL.
AI_NUMERICHOST	Specifies that the *`hostname`* value is a numeric address string. If this flag is set and `hostname` is not a numeric address string, the returned value is set to EAI_NONAME. Use this flag to prevent calling a name resolution service like DNS.
AI_NUMERICSERV	Specifies that the service value is a non NULL numeric port string. If this flag is set and service is not a numeric port string, the returned value is set to `EAI_NONAME`. Use this flag to prevent calling a name resolution service like DNS.
AI_V4MAPPED	Requests the return of all IPv4-mapped IPv6 addresses when the address family is AF_INET6 and no matching IPv6 addresses exist. This flag is ignored if the address family is AF_INET.
AI_ALL	Requests the return of all matching IPv4 and IPv6 records. This flag is ignored unless `AI_V4MAPPED` is also set.
AI_ADDRCONFIG	Requests the return of only IPv6 records if a host has at least one IPv6 source address configured, or only IPv4 records if a host has at least one IPv4 source address configured.
AI_DEFAULT (AI_V4MAPPED | AI_ADDRCONFIG)	If AI_ADDRCONFIG | AI_V4MAPPED is specified, the A records are returned as IPv4-mapped IPv6 addresses.

If no error is returned, points to a list of addrinfo structs. For each addrinfo struct, ai_family, ai_socktype, and ai_protocol may be used as arguments to the socket function.

`ai_family`
indicates a literal of the form `PF_xxx`, where `xxx` indicates the address family as a protocol family name. This member can be used with the `socket` function.

`ai_socktype`
indicates a literal of the form `SOCK_xxx`, where *`xxx`* indicates the socket type.

`ai_protocol`
indicates either 0 (zero) or a literal of the form `IPPROTO_xxx`, where *`xxx`* indicates the protocol type.

`ai_addrlen`
is the length of the socket address.

`ai_canonname`
is a pointer to the canonical name for the host.

`ai_addr`
is a pointer to the socket address structure that can be used with any socket function that requires a socket address. The length of a specific `ai_addr` value is described by the member named `ai_addrlen`.

`ai_next`
is a pointer to the next structure in the linked list.

arpreq

The ARP request structure is used by Address Resolution Protocol (ARP) I/O control operations. This structure is defined in the `ifarp.h` header file. Use this structure in applications that assume some of the functions of a transport protocol such as TCP or UDP, or bootp.

`C Declaration`

```
#include <ifarp.h>
struct arpreq {
```

```
        struct  sockaddr arp_pa;
        struct {
                 unsigned short   sa_family;
                 unsigned char    sa_data[6];
        }arp_ha;
        short    arp_flags;
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT arpha (*);
        BEGIN
                INT     sa_family;
                STRING  sa_data[0:5];
        END;

STRUCT arpreq;
      BEGIN
            STRUCT      arp_pa (sockaddr);
            STRUCT      arp_ha (arpha);
            INT         arp_flags;
      END;
```

arp_pa
contains the Internet address of the machine.

NOTE: Since `arp_pa` is a `sockaddr` struct, it contains fields for the port, address family, and Internet address. However, ARP is only concerned with the Internet address. The programmer is responsible for filling the port and address family fields with null values.

sa_family
is the type of address. Its value is always `AF_UNSPEC`.

sa_data
contains the Ethernet address of the machine specified in *arp_pa*.

arp_flags
contains a combination of one or more of the following flags:

`ATF_INUSE`	Indicates the entry is in use.
`ATF_COM`	Indicates a completed entry (the Ethernet address is valid).
`ATF_PERM`	Indicates a permanent entry.
`ATF_PUBL`	Indicates a publish entry (that is, a response for another host).

hostent

This structure is used by the support routines to hold hostname and address information. It is defined in the `netdb.h` header file.

C Declaration

```
#include <netdb.h>
struct hostent   {
        char     *h_name;
        char    **h_aliases;
        int       h_addrtype;
        int       h_length;
        char    **h_addr_list;
#define h_addr    h_addr_list[0]
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT haliase (*);

      BEGIN
           STRING  .EXT ptrs;
      END;

STRUCT hptrs (haliase)[0:4];

STRUCT ha_aliase (*);

      BEGIN
            STRING  .EXT ptrs;
      END;

STRUCT ha_ptrs (ha_aliase)[0:4];

STRUCT hostent  (*);
      BEGIN
            STRING      .EXT h_name;
            STRING      .EXT h_aliases (hptrs);
            INT(32)          h_addrtype;
            INT(32)          h_length;
            STRING      .EXT h_addr_list (ha_ptrs);
      END;
```

h_name
points to the official name of the host.

h_aliases
points to an array of pointers to the various aliases assigned to the host.

h_addrtype
is the type of address. Its value is always AF_INET, indicating an Internet address.

h_length
is the length, in bytes, of each entry pointed to by *h_addr_list*. Usually, the length is 4 bytes.

h_addr_list
points to an array of null-terminated pointers to the addresses from the name server, in network order.

if_nameindex

The name index structure holds information for a single interface. This structure is defined in the if.h header file. The if_nameindex function returns an array of if_nameindex structures with one structure for each interface. The if_freenameindex function frees the memory used for this array of structures. This structure applies to NonStop TCP/IP only.

C Declaration

```
#include <if.h>
struct  if_nameindex {

        unsigned int     if_index;
        char            *if_name;
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
```

```
STRUCT if_nameindex_tal (*);

      BEGIN
          INT(32)       if_index;
          STRING .EXT   if_name;
      END;
```

if_index
specifies the index to be mapped to an interface name.

if_name
specifies the buffer to receive the mapped name. The buffer must be at least `IF_NAMESIZE` bytes long; `IF_NAMESIZE` is defined in the header file `in.h`.

ifreq

The interface request structure is used for socket I/O control operations. All interface control operations must have parameter definitions that begin with `ifr_name`. The remaining definitions can be interface-specific. This structure is defined in the `if.h` header file. Use this structure if you are writing a transport protocol such as TCP.

C Declaration

```
#include <if.h>
struct  ifreq {
#if defined(_GUARDIAN_TARGET) \\ defined (_GUARDIAN_SOCKETS)
        unsigned long    ifr_filler;
#endif
#define IFNAMSIZ         16
        char             ifr_name[IFNAMSIZ];
        union {
                struct   sockaddr ifru_addr;
                struct   sockaddr ifru_dstaddr;
                struct   sockaddr ifru_broadaddr;
                short    ifru_flags;
                int      ifru_metric;
                caddr_t  ifru_data;
                int      ifru_value;
                u_long   ifru_index;
        } ifr_ifru;
#define ifr_addr         ifr_ifru.ifru_addr
#define ifr_dstaddr      ifr_ifru.ifru_dstaddr
#define ifr_broadaddr    ifr_ifru.ifru_broadaddr
#define ifr_flags        ifr_ifru.ifru_flags
#define ifr_metric       ifr_ifru.ifru_metric
#define ifr_data         ifr_ifru.ifru_data
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT ifreq (*);

      BEGIN
            INT(32)     ifr_filler;
            STRING      ifr_name [0:IFNAMESIZ-1];
            STRUCT      ifr_addr (sockaddr);
            STRUCT      ifr_dstaddr (sockaddr)    = if_addr;
            STRUCT      ifr_broadaddr (sockaddr) = if_addr;
            INT(32)     ifr_flags                 = if_addr;
            STRING .EXT ifr_metric                = if_addr;
      END;
```

ifr_name[IFNAMESIZ]
contains the name of the SUBNET device. The name must begin with the pound sign (#), followed by the interface name in all capital letters.

ifr_addr
is the interface address.

ifr_dstaddr
is the destination address at the other end of a point-to-point link.

ifr_broadaddr
is the broadcast address of this interface.

ifr_flags
contains a combination of one or more of the following flags:

IFF_UP	Indicates that the interface is up.
IFF_BROADCAST	Indicates that this is a broadcast-oriented interface (such as Ethernet).
IFF_LOOPBACK	Indicates that this is a loopback interface.
IFF_POINTTOPOINT	Indicates that this is a point-to-point link.
IFF_RUNNING	Indicates that the interface is active.
IFF_NOARP	Indicates that the interface does not support ARP.

ifr_metric
gets or sets the interface metric; it is used by routing programs. Refer to the *TCP/IP Configuration and Management Manual* for details on routing.

ifr_data
contains the data associated with the request.

ifr_value
is any generic value.

ifr_index
is an interface index.

in_addr

This is a 4-byte structure that defines an Internet address. This structure is used by the socket routines and is declared in the `in.h` header file.

C Declaration

```
#include <in.h>
struct in_addr  {
        in_addr_t    s_addr;
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT in_addr (*);
        BEGIN
                INT(32)      s_addr;
        END;
```

s_addr
is the Internet address.

in6_addr

This structure holds a single IPv6 address. This structure is implemented with an embedded union with extra fields that force an alignment level in a manner similar to BSD implementations of struct in_addr. This structure is used by the socket routines and is declared in the `in6.h` header file. This structure applies to NonStop TCP/IP only.

C Declaration

```
#include <in6.h>
struct in6_addr union {
        u_char   sa6_addr[16];
#define s6_addr         s6_un.sa6_addr
        u_short  sa6_waddr[8];
#define s6_waddr        s6_un.sa6_waddr
        u_long   sa6_laddr[4];
#define s6_laddr        s6_un.sa6_laddr
#ifdef IN6_HAS_64BIT_INTTYPE
        uint64_t sa6_qaddr[2];
#define s6_qaddr        s6_un.sa6_qaddr
#endif
    } s6_un;
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT .in6_addr;
        BEGIN
            STRING  s6_addr[0:15];       !128-bit IPv6 addr
            INT     s6_waddr = s6_addr;  !as 8 words
            INT(32) s6_laddr = s6_addr;  !as 4 longs
            FIXED   s6_qaddr = s6_addr;  !as 2 quads
        END;
```

sa6_addr[16]
: a host address formatted as 16 u_chars.

sa6_waddr[8]
: a host address formatted as eight u_shorts.

sa6_laddr
: a host address formatted as four u_longs.

sa6_qaddr
: a host address formatted as two uint64_ts.

ip_mreq

The IP multicast request structure is used for multicast socket I/O control operations. This structure is used by the socket routines and is declared in the `in.h` header file

C Declaration

```
#include <in.h>
struct ip_mreq  {

      struct in_addr imr_multiaddr; /* IP multicast group
                                       address */
      struct in_addr imr_interface; /* local interface IP
                                       address */
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT ip_mreq (*);

   BEGIN
    STRUCT imr_multiaddr (in_addr); !IP multicast group address
                                    !local interface
    STRUCT imr_interface (in_addr); !IP address
   END;
```

imr_multiaddr
contains the address of the IP multicast group to join membership to or drop membership from.

imr_interface
is the interface IP address.

ipv6_mreq

The IP multicast request structure is used for IPv6 multicast socket I/O control operations. This structure is used by the socket routines and is declared in the `in6.h` header file. This structure applies to NonStop TCP/IP only.

C Declaration

```
#include <in6.h>
struct ipv6_mreq  {

   struct in6_addr ipv6mr_multiaddr; /* IPv6 multicast address */
   unsigned int    ipv6mr_interface; /* interface index */

};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT .ipv6_mreq;

   BEGIN
    STRUCT sin6_addr(in6_addr);      !IPv6 address
    INT(32) ipv6mr_interface;        !local interface
   END;
```

ipv6mr_multiaddr
contains the address of the IPv6 multicast group to join membership to or drop membership from. Can be specified with a value of 0, which allows an application to choose the default multicast interface.

ipv6mr_interface
is the local interface IPv6 address.

netent

This structure is used by the support routines that deal with network names. It is defined in the `netdb.h` header file. This structure is used by the `getnetbyname` and `getnetbyaddr` support routines.

C Declaration

```
#include <netdb.h>
struct netent    {
        char           *n_name;
        char           **n_aliases;
        int            n_addrtype;
```

```
        unsigned long  n_net;
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT naliase (*);

    BEGIN
      STRING  .EXT ptrs;
    END;

STRUCT nptrs (naliase)[0:4];

STRUCT netent (*);
      BEGIN
        STRING  .EXT n_name;
        STRING  .EXT n_aliases(nptrs);
        INT(32)      n_addrtype;
        INT(32)      n_net
END;
```

n_name
: points to the official name of the network.

n_aliases
: points to an array of null-terminated pointers to various aliases for the network.

n_addrtype
: indicates the type of network number returned; its value is always AF_INET, indicating the network part of an Internet address.

n_net
: is the network number, in host order.

open_info_message

This structure is used by the routines that deal with obtaining information for the primary and backup processes of a NonStop process pair. It is defined in the netdb.h header file. This structure is used by the socket_get_open_info and socket_backup routines. Additional information about the parameters for this structure can be found in the description of the FILE_OPEN_ procedure in the *Guardian Procedure Calls Reference Manual*.

C Declaration

```
#include <netdb.h>
struct open_info_message   {
         short   filenum;
         char    file_name[32];
         short   filename_len;
         short   flags;
         short   sync;
         short   access;
         short   exclusion;
         short   nowait;
         short   options;
        };
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT open_info_message (*);
    BEGIN
         INT     filenum;
         STRING  file_name[0:31];
```

```
        INT     filename_len;
        INT     flags;
        INT     sync;
        INT     access;
        INT     exclusion;
        INT     nowait;
        INT     options;
    END;
```

filenum
specifies the file number of the opened file.

file_name
is the name of the file.

filename_len
is the length, in bytes, of the contents of *file_name*.

flags
specifies flag values that affect the file.

sync
specifies the sync-depth value of the file.

access
is the access mode of the file.

exclusion
is the mode of compatibility with other openers of the file.

nowait
defines whether I/O operations for the file are to be nowait operations.

options
is the optional characteristics of the file.

protoent

This structure is used by the support routines that deal with protocol names. This structure is defined in the `netdb.h` header file.

C Declaration

```
#include <netdb.h>

struct protoent   {
        char      *p_name;
        char     **p_aliases;
        int        p_proto;
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT paliase (*);

      BEGIN
           STRING  .EXT ptrs;
      END;

STRUCT pptrs (paliase)[0:4];

STRUCT protoent (*);

      BEGIN
```

```
            STRING      .EXT p_name;
            STRING      .EXT p_aliases(pptrs);
            INT(32)      p_proto;
      END;
```

p_name
points to the official name of the protocol.

p_aliases
points to an array of null-terminated pointers to various aliases for the protocol.

p_proto
is the protocol number.

rtentry

The route entry structure is used when adding or deleting routes. It is defined in the `route.h` header file. NonStop TCP/IPv6 and NonStop TCP/IPv6 distinguish between routes to hosts and routes to networks. When available, routes to hosts are preferred.

The interface to be used for each route is inferred from the gateway address supplied when the route is entered. Routes that forward packets through gateways are marked so output routines can determine that the packets are routed through a gateway, rather than directly to the destination host.

C Declaration

```
#include <route.h>
#define RT_MAXNAMESIZ 12

struct rtentry {
        unsigned long   rt_hash;
        struct sockaddr rt_dst;
        struct sockaddr rt_gateway;
        short           rt_flags;
        short           rt_refcnt;
        unsigned long   rt_use;
        struct          ifnet *rt_ifp;
#ifdef __TANDEM
        double          rt_resettime;
        unsigned char   rt_name[RT_MAXNAMESIZ];
        ushort          context_val;
#endif /* __TANDEM */
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT rtentry (*);
      BEGIN
            INT(32)      rt_hash;
            struct       rt_dst (sockaddr);
            struct       rt_gateway (sockaddr);
            INT          rt_flags;
            INT          rt_refcnt;
            INT(32)      rt_use;
            INT(32)     .rt_ifp (ifnet);
            REAL(64)     rt_resettime;
            STRING       rt_name[0:RT_MAXNAMESIZ-1];
      END;
```

rt_hash
is not used.

rt_dst
is the destination of the route.

rt_gateway
is the gateway to the destination.

rt_flags
contains a combination of one or more of the following flags:

RTF_UP	Indicates the route is up and can be used.
RTF_GATEWAY	Indicates the destination is a gateway.
RTF_HOST	Indicates the route is a host entry in a point-to-point table. (Otherwise, the route is an entry in a network table.)
RTF_MDOWN	Indicates the route has been temporarily marked down.
RTF_DYNAMIC	Indicates the route was created dynamically; that is, by redirection of an Internet Control Message Protocol (ICMP) route.

rt_refcnt
is not used.

rt_use
is not used.

rt_ifp
is not used.

rt_resettime
is not used.

rt_name
is not used.

context_val
is not used.

send_nw_str

This structure is used by the send_nw routine. It is defined in the `netdb.h` header file.

C Declaration

```
#include <netdb.h>
struct send_nw_str {
        int  nb_sent;
        char nb_data[1];
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT send_nw_str(*);

      BEGIN
            INT(32)     nb_sent;
            STRING      nb_data [0:1];
      END;
```

nb_sent
is the number of bytes sent by the send operation.

nb_data[1]
is the first character of the data to be sent.

sendto_recvfrom_buf

This structure is used by the recvfrom_nw and sendto_nw routines. It is defined in the `in.h` header file.

C Declaration

```
#include <in.h>
struct sendto_recvfrom_buf   {
        struct  sockaddr_in sb_sin;
        char                sb_data[1];
};
#define  sb_sent     sb_sin.sin_family
#define SOCKADDR_IN
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT sendto_recvfrom_buf (*);

      BEGIN
            STRUCT      sb_sin(sockaddr_in);
            STRING      sb_data[0:1];
      END;
```

sb_sin
is an address-port number combination based on the structure `sockaddr_in`.

sb_data
provides a symbolic name that can be used to locate the start of the user data.

sb_sent
is the number of bytes that have been transferred by a call to the `t_sendto_nw` function (followed by a call to the `AWAITIOX` procedure). Check this value after the `AWAITIOX` call completes.

servent

This structure is used by the support routines to convert service names to port numbers. It is defined in the `netdb.h` header file. Use this structure if you are writing a network service manager similar to the HP NonStop LISTNER process or the UNIX inetd daemon.

C Declaration

```
#include <netdb.h>

struct servent   {
        char     *s_name;
        char    **s_aliases;
        int       s_port;
        char     *s_proto;
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT aliase (*);
      BEGIN
           STRING  .EXT ptrs;
      END;

STRUCT sptrs(aliase)[0:3];

STRUCT servent (*);
      BEGIN
            STRING   .EXT s_name;
```

```
            STRING    .EXT s_aliases(sptrs);
            INT(32)        s_port;
            STRING    .EXT s_proto;
      END;
```

s_name
points to the official name of the service.

s_aliases
points to an array of null-terminated strings to the various aliases for the service.

s_port
is the port number associated with the service, in network order.

s_proto
points to the name of the protocol associated with the service.

sockaddr

This structure, defined in the `in.h` header file, is a pointer to the `sockaddr_in` structure.

C Declaration

```
#include <in.h>
struct sockaddr     {
        sa_family_t     sa_family;
        char            sa_data[SA_DATA_SIZE];
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT sockaddr (*);
      BEGIN
            INT          sa_family;
            STRING       sa_data[0:SA_DATA_SIZE-1];
      END;
```

sa_family
is the address family.

sa_data
contains up to 14 bytes of direct address.

Usage Guidelines

This structure makes the HP NonStop Kernel Operating System User's Guide, Parallel Library TCP/IP, and NonStop TCP/IP subsystems compatible with other implementations. When you pass a parameter of this type to a socket routine, the fields filled or read are those of the sockaddr_in structure.

For example, consider the following program excerpts:

```
#include "$system.ztcpip.inh"
...
struct sockaddr_in sin;
...
s1 = socket (AF_INET, SOCK_STREAM, 0);
...
sin.sin_family = AF_INET; /* 2 byte short int */
sin.sin_addr.s_addr = INADDR_ANY; /* 4 byte Internet addr */
sin.sin_port = port; /* 2 byte short int */
bind (s1, (struct sockaddr *)&sin, sizeof (sin));
```

The #include directive contains the declaration of the sockaddr_in structure. The program declares that the sin structure is based on the sockaddr_in structure. The socket s1 is created by a call to the socket routine. The bind routine syntax requires that the address and port number that you want to bind to the socket be stored in a structure based on the sockaddr_in structure. The routine also requires that you pass a pointer to that structure (sin, in this example).

The following program excerpt shows an example for IPv6:

```
#include "$system.ztcpip.in6h"
..

struct sockaddr_in6 sin;
s1= socket (AF_INET6, SOCK_STREAM, 0);
sin6.sin6.family=AF_INET6;
sin6.sin6_port;
sin6.sin6_addr=in6addr_any;

bind (s1,struck sockaddr *)&sin, sizeof(sin));
```

sockaddr_in

This structure defines an address-port number combination that is used by many of the socket routines. It is defined in the in.h header file.

C Declaration

```
#include <in6.h> struct sockaddr_in     {
         short             sin_family;
         unsigned short    sin_port;
         struct  in_addr   sin_addr;
         char              sa_data[8]; };
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT sockaddr_in (*);
      BEGIN
            INT         sin_family;
            INT         sin_port;
            STRUCT      sin_addr(in_addr);
            STRING      sa_data[0:8];
      END;
```

sin_family
is the type of address. Its value is always AF_INET because only Internet addresses are supported.

sin_port
is the port number associated with the socket.

sin_addr
is the Internet address (based on the in_addr structure) associated with the socket.

sa_data
is not currently used. It is reserved for future use.

sockaddr_in6

This structure specifies a local or remote endpoint address to which to connect a socket. This structure is IPv6 specific and is defined in the in6.h header file. This structure applies to NonStop TCP/IPv6 only.

C Declaration

```
#include <in6.h>
```

```
struct sockaddr_in6  {
     u_short          sin6_family; /* AF_INET6 */
     u_short          sin6_port; /* Transport layer port # */
     u_long           sin6_flowinfo; / *IPv6 flow info */
     struct  in6_addr sin6_addr; /* IPv6 address */
     u_long           sin6_scope_id; / *set of interfaces for scope */
};
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT sockaddr_in6 (*);
      BEGIN
            INT          sin6_family;
            INT          sin6_port;
            INT(32)      sin6_flowinfo;
      STRUCT             sin6_addr(in6_addr);
            INT(32)      sin6_scope_id;
      END;
```

sin6_family
is the type of address. Its value is always AF_INET6.

sin6_port
is the port number associated with the socket.

sin6_flowinfo
is the flow label value.

sin6_addr
is the Internet address (based on the in6_addr structure) associated with the socket.

sin6_scope_id
is the set of interfaces that are associated with the scope.

sockaddr_storage

This structure defines an IPv6 address-port number combination that is used by many of the socket routines. This structure is defined in the socket.h header file. This structure applies to NonStop TCP/IP only.

C Declaration

```
#include <socket.h>
struct sockaddr_storage  {
     sa_family_t   __ss_family;
     char          __ss_pad1[_SS_PAD1SIZE];
     int64_t       __ss_align;
     char          __ss_pad2[_SS_PAD2SIZE]; };
```

TAL Declaration

```
?NOLIST, SOURCE SOCKDEFT
STRUCT sockaddr_storage (*);

      BEGIN
         INT           _ss_family;
         STRING        _ss_pad1[0:5];
         FIXED         _ss_align;
         STRING        _ss_pad2[0:111];
      END;
```

__ss_family
is the address family.

__ss_pad1
is a 6-byte pad up to the _ss_align field.

__ss_align
forces the alignment of the field.

__ss_pad2
is the 112-byte pad to the desired size of the field.

4 Library Routines

This section contains the syntax and semantics for the socket-library routines provided by the NonStop TCP/IP, NonStop TCP/IP, and NonStop TCP/IP products. These routines are compatible with the socket routines in the 4.3 BSD UNIX operating system, except as noted here or in the Porting Considerations (page 32).

In addition to the sockets library, which is implemented in the C language, NonStop TCP/IPv6, NonStop TCP/IP, and NonStop TCP/IP provide a TAL binding to the sockets library to support applications written in TAL.

Where this section documents library routines that are only available for the NonStop TCP/IP subsystem, it is indicated in the description of the routine.

Socket Library Routines

The socket library routines are provided in two sets of three files each. One set is Common Run-Time Environment (CRE) dependent (CRE-dependent) and the other set has no dependence on CRE (CRE-independent). See CRE Considerations (page 88) for more information about CRE.

For enabling 64–bit features, call 64–bit APIs in the application and recompile with 'lp64' complier option.

CRE-Dependent Socket Library

The CRE-dependent socket library is neutral with respect to the Common Runtime Environment (CRE), in that it uses no routines that depend on CRE; however, this library does depend on CRE for the global *errno* data variable which permits applications to use the perror function. The CRE-dependent, non-native, socket library routines are provided in two versions for data storage: one for the large-memory model and one for the wide-data model.

The large-memory-model routines are in the file `$SYSTEM.ZTCPIP.LIBINETL`. The wide-data-model routines are in `$SYSTEM.ZTCPIP.LIBINETW`. TAL routines are provided by the prototype procedures contained in SOCKPROC.

Native C users should use the SRL version of the socket library, `ZINETSRL`.

Current users of the wide-data-model routines, LIBINETW, require no changes to their application code to utilize the D40-native socket library. These applications must, however, be recompiled using the D40 header files.

Applications using the large-memory-model routines, LIBINETL, need to verify that the correct data types are used in function calls to the socket library. If the correct data types are specified, the only requirement is a recompilation using the D40 header files. Otherwise, the data types must be changed to reflect the function descriptions in this manual.

Refer to the *C/C++ Programmer's Guide* for more details on memory models.

CRE-Independent Socket Library

The CRE-Independent socket library routines are provided in three versions for data storage. Two are non-native versions, one for the large-memory model and one for the wide-data model. The large-memory-model routines are in the file `$SYSTEM.ZTCPIP.LNETINDL`. The wide-data-model routines are in `$SYSTEM.ZTCPIP.LNETINDW`. The native-linkable version is in the file LNETINDN.

Refer to the *C/C++ Programmer's Guide* for more details on memory models.

Summary of Routines

Both sets of the socket library contain two main types of routines: socket routines and support routines.

Socket routines deal directly with connections and data transfer.

Support routines assist in name translation, enabling you to use easy-to-understand symbolic names for objects, hosts, and services. However, they are not essential for data transmission using the socket library, and only two of them—`gethostname` and `gethostid`—communicate with the TCP/IP process.

NOTE: Certain socket options are supported differently in CIP. See the *Cluster I/O Protocols (CIP) Configuration and Management Manual* for details.

Table 12 lists and briefly describes each socket routine and provides the page number where the routine is described.

Table 12 Socket Routines

Name and Description Page	Function
accept (page 89)	Listens for connections on an existing socket, creates a new socket for data transfer, and accepts a connection on the new socket (waited)
accept_nw (page 91)	Listens for connections on an existing socket (nowait)
accept_nw1 (page 94)	Allows you to change queue length when listening for connections on an existing socket (nowait)
accept_nw2 (page 95)	Creates a new socket for data transfer and accepts a connection on the new socket (nowait)
bind, bind_nw (page 98)	Binds a socket to an address and port number (waited or nowait)
connect, connect_nw (page 102)	Connects a socket to a remote socket (waited or nowait)
getsockname, getsockname_nw (page 126)	Gets the address and port number to which a socket is bound (waited or nowait)
getsockopt, getsockopt_nw (page 128)	Gets socket options (waited or nowait)
if_freenameindex (page 130)	Sets the queue length (provided for compatibility only; queue length always set to 5)
recv, recv_nw (page 153)	Receives data on a socket (waited or nowait)
recv64_, recv_nw64_ (page 155)	Receives data on a socket (waited or nowait) in 64–bit application.
recvfrom (page 158)	Receives data on an unconnected UDP or raw socket (waited and nowait)
recvfrom64_	Receives data on an unconnected UDP or raw socket (waited and nowait) in 64–bit application.
recvfrom_nw (page 161)	Receives data on an unconnected UDP socket or raw socket created for nowait operations
recvfrom_nw64_ (page 164)	Receives data on an unconnected UDP socket or raw socket created for nowait operations in 64–bit application.
send (page 166)	Sends data on a socket (waited)
send64_ (page 168)	Sends data on a socket (waited) in 64–bit application.
send_nw (page 169)	Sends data on a socket (nowait)
“send_nw64_” (page 171)	Sends data on a socket (nowait) in 64–bit application.
send_nw2 (page 173)	Sends data on a socket without byte-count header (nowait)
send_nw2_64_ (page 175)	Sends data on a socket without byte-count header (nowait) in 64–bit application.
sendto (page 177)	Sends data on an unconnected UDP or raw socket (waited)
sendto64_ (page 179)	Sends data on an unconnected UDP or raw socket (waited) in 64–bit application.

Table 12 Socket Routines *(continued)*

Name and Description Page	Function
sendto_nw (page 180)	Sends data on an unconnected UDP or raw socket without byte-count header (nowait)
sendto_nw64_ (page 182)	Sends data on an unconnected UDP or raw socket without byte-count header (nowait) in 64–bit application.
setsockopt, setsockopt_nw (page 184)	Sets socket options (waited and nowait)
shutdown, shutdown_nw (page 189)	Shuts down data transfer on a socket (waited or nowait)
sock_close_reuse_nw (page 190)	Marks the socket for reuse
socket, socket_nw (page 191)	Creates a socket (waited or nowait)
socket_backup (page 193)	Allows an application to establish a backup TCP/IP process
socket_get_info (page 194)	Obtains address and length of data received from an unconnected UDP or raw socket
socket_get_len (page 195)	Obtains byte count of data sent on a socket
socket_get_open_info (page 196)	Obtains parameters used to open a TCP/IP process. Used to checkpoint* information for NonStop process pairs.
socket_ioctl, socket_ioctl_nw (page 197)	Performs a control operation on a socket (waited or nowait)
socket_set_inet_name (page 200)	Sets the name of the NonStop TCP/IPv6, TCPSAM, or TCP6SAM process

*"Checkpoint" here refers to sending state-change information from the primary to the backup process.

Table 13 (page 83) lists and briefly describes each of the support routines. All of the support calls are waited calls.

Table 13 Support Routines

Routine Name	Functions
freeaddrinfo (page 104)	Frees a specified address-information structure previously created by the `getaddrinfo` function. (Supported by NonStop TCP/IPv6 only.)
freehostent (page 105)	Frees the memory of one or more `hostent` structures returned by the `getipnodebyaddr` or `getipnodebyname` functions. (Supported by HP NonStop Kernel Operating System User's Guide only.)
gai_strerror (page 105)	Aids applications in printing error messages returned by `getaddrinfo`. (Supported by NonStop TCP/IP only.)
getaddrinfo (page 107)	Converts hostnames and service names into socket-address structures. (Supported by NonStop TCP/IPv6 only.)
gethostbyaddr, host_file_gethostbyaddr (page 109)	Gets the Internet address of the specified host.
gethostbyname, host_file_gethostbyname (page 110)	Gets the name of the host with the specified Internet address.
gethostbyname2 (page 112)	Gets the Internet address (IPv4 or IPv6) of the host whose name is specified.
gethostid (page 113)	Gets the ID of the current host.
gethostid (page 113)	Gets the ID of the current host.
gethostname (page 113)	Gets the name of the current host.

Table 13 Support Routines *(continued)*

Routine Name	Functions
getipnodebyaddr (page 114)	Gets the name of the host that has a specified Internet address and provides an error-number value to maintain a thread-safe environment. (Supported by NonStop TCP/IP only.)
getipnodebyname (page 116)	Provides lookups for IPv4/IPv6 hosts. (Supported by NonStop TCP/IPv6 only.)
getnameinfo (page 117)	Translates a protocol-independent host address to a hostname and gives the service name. (Supported by NonStop TCP/IPv6 only.)
getnetbyaddr (page 119)	Gets the name of the network with the specified network address.
getnetbyname (page 120)	Gets the Internet address of the network with the specified name
getprotobyname (page 122)	Gets the protocol with the specified name
getprotobynumber (page 123)	Gets the protocol with the specified protocol number
getservbyname (page 124)	Gets the service port number for a given service name
getservbyport (page 125)	Gets the service name for a given port number
if_freenameindex (page 130)	Frees dynamic memory allocated by the `if_nameindex` function. (Supported by NonStop TCP/IPv6 only.)
if_indextoname (page 131)	Maps an interface index to its corresponding name. (Supported by NonStop TCP/IPv6 only.)
if_nameindex (page 132)	Gets all interface names and indexes. (Supported by NonStop TCP/IPv6 only.)
if_nametoindex (page 133)	Maps an interface name to its corresponding index. (Supported by NonStop TCP/IP only.)
inet_addr (page 134)	Converts an Internet address from dotted-decimal format to binary format
inet_lnaof (page 135)	Breaks apart an Internet address and returns the local address portion
inet_makeaddr (page 135)	Combines a network address and a local address to create an Internet address
inet_netof (page 136)	Breaks apart an Internet address and returns the network address portion
inet_network (page 136)	Converts an Internet address from dotted-decimal format to binary format and returns the network address portion
inet_ntoa (page 137)	Converts an Internet address from binary format to dotted-decimal format
inet_ntop (page 138)	Converts a binary IPv6 or IPv4 address to a character string. (Supported by Parallel Library TCP/IP only.)
inet_pton (page 139)	Converts a character string to a binary IPv6 or IPv4 address. (Supported by NonStop TCP/IPv6 only.)
lwres_freeaddrinfo (page 140)	Frees the memory of one or more `addrinfo` structures previously created by the `lwres_getaddrinfo` function. (Supported by NonStop TCP/IP only.)
lwres_freehostent (page 141)	Frees the memory of one or more `hostent` structures returned by the lwres_getipnodebyaddr or lwres_getipnodebyname functions. (Supported for NonStop TCP/IPv6 only.)
lwres_gai_strerror (page 141)	Aids applications in printing error messages based on the EAI_ codes returned by the lwres_getaddrinfo function. (Supported for NonStop TCP/IPv6 only.)

Table 13 Support Routines *(continued)*

Routine Name	Functions
lwres_getaddrinfo (page 142)	Converts hostnames and service names into socket address structures. (Supported for NonStop TCP/IPv6 only.)
lwres_gethostbyaddr (page 144)	Gets the name of the host that has the specified Internet address and address family. (Supported for Parallel Library TCP/IP only.)
lwres_gethostbyname (page 145)	Gets the Internet address (IPv4) of the host whose name is specified. (Supported for NonStop TCP/IPv6 only.)
lwres_gethostbyname2 (page 146)	Gets the Internet address (IPv4 or IPv6) of the host whose name is specified. (Supported for NonStop TCP/IPv6 only.)
lwres_getipnodebyaddr (page 147)	Searches host entries until a match with src is found. (Supported for NonStop TCP/IPv6 only.)
lwres_getipnodebyname (page 149)	Gets host information based on the hostname. (Supported for NonStop TCP/IPv6 only.)
lwres_getnameinfo (page 150)	Translates a protocol-independent host address to a hostname. (Supported for NonStop TCP/IPv6 only.)
lwres_hstrerror (page 152)	Returns an appropriate string for the error code given by err_num. (Supported for NonStop TCP/IPv6 only.)

Syntax and Semantics of Socket Library Routines

This subsection describes each routine in the socket library. The routines are listed alphabetically. Each description includes the following information:

- What the routine does
- What headers you need to specify in an `#include` statement within your programs before calling the routine
- What arguments the routine accepts and how it interprets them
- What value the routine returns and how you should interpret it
- What types you must declare for each argument and for the return value
- What errors can be returned
- What guidelines you need to consider when using the routine

Many of the descriptions include an example that shows how to use the routine.

See Chapter 3 (page 62) for a summary of the C header files provided with the socket library and for descriptions of the data structures provided in the header files.

All return codes and values are of type integer unless otherwise noted.

Nowait Routines

Most of the socket routines have two versions: one for waited operations and another for nowait operations. The names of the nowait routines end in the suffix `_nw`. Except for the `socket_nw` routine, the nowait routines include an additional *`tag`* parameter that is passed to the NonStop operating system file-system procedures.

Error Conditions

Most routines that refer to a socket number (*`socket`*), plus a few support routines, indicate an error condition by returning an otherwise impossible return value (usually -1) and placing the appropriate error number in the external variable *`errno`*. Since *`errno`* is not cleared on successive calls, you should test it only after an error has occurred. You can call the perror function to print

the text message associated with the current error to the standard C error file (the file named stderr).

The text message description of each routine lists most error numbers that can be returned in *errno* on a call to the particular routine. A complete list of socket errors and their meanings is given in Appendix B (page 243). You must interpret the meaning of each error according to the type of call and the circumstances in which your program issues the call. For more information, see Asynchrony and Nowaited Operations (page 34).

NOTE: The perror function is not supported for TAL sockets.

Nowait Call Errors

The nowait versions of the routines return an error in the file-system variable. Call FILE_GETINFO_ procedure after calls to either AWAITIOX or FILE_AWAITIO64_ to get the error. You must set this error in the *errno* variable in the application.

NOTE: When you initiate a nowait call, *errno* is set to reflect any error detected upon initiation. If *errno* is nonzero after initiation, your program should not call the AWAITIOX procedure because the operation is not successfully initiated.

Socket error numbers are in the range reserved by the NonStop operating system for application-defined errors. These do not conflict with the range the operating system has reserved for file-system errors. However, it is possible to get regular NonStop operating system file-system errors that pertain to interprocess I/O, because the socket routines are built on NonStop operating system interprocess I/O. For descriptions of these interprocess I/O errors, refer to the *Guardian Procedure Calls Reference Manual*.

The gethostbyname, gethostbyaddr, host_file_gethostbyname, and host_file_gethostbyaddr support routines indicate an error value in another external variable, *h*_errno. If you bypass the Domain Name resolver code, the only possible nonzero (error) value of *h*_errno is HOST_NOT_FOUND(1). If you use the resolver code, four error values are possible. These errors are described with the functions gethostbyaddr and gethostbyname, in this section.

Interfacing TAL Programs to the Socket Library

NOTE: For more information about socket library support for TAL and pTAL applications, see TALDOCUM in $system.ztcpip.

A program is considered a TAL program if its MAIN section is written in TAL. A program that has a C main section but calls TAL procedures is not bound by the requirements given in this subsection. The topics covered include:

- Implications of the C socket library
- Startup considerations
- Bind considerations
- CRE considerations

Any experience writing C language socket code is applicable to writing TAL socket code. All the functions, parameters, data structures, and return codes are the same in TAL as they are in C. The differences are only a matter of TAL syntax.

NOTE: Use the Common Run-Time Environment (CRE) when developing TAL socket applications. CRE is described in the *CRE Programming Manual*.

Procedure Prototypes

Each socket function described in this manual is available to be “sourced” into TAL programs. Either the entire set of prototypes or individual functions may be sourced.

Because TAL procedures cannot be type cast for returning pointers, those procedures that actually do return pointers are typed as `INT(32)`. It is the programmer’s responsibility to redefine the returned `INT(32)` as a pointer to the appropriate structure. It may be helpful for the TAL programmer to think of these pointers to structures as pointers to arrays.

Implications of the C Socket Library

TAL programs bound with the socket library differ significantly from applications written completely in C. TAL programs miss the normal C run-time library and the normal startup logic. The full C run-time library is replaced by a subset of minimal functions that are used by the socket library. This means that a programmer who wishes to combine C and TAL procedures to implement an application is bound by this same minimal C run-time library functionality.

The TAL version of the socket library is based on the C large-memory model, so all pointers must be 4-byte pointers.

The pTAL version of the socket library is based on the C wide-data model, so all pointers must be 4-byte pointers.

The functions provided include:

- Very minimal `STDIO` functionality:
  - fopen
  - fgets
  - fclose
- `'str...'` functionality
- `'mem...'` functionality
- `sprintf`, but not `fprintf` or `printf`
- All functions implemented as macros
- *`errno`* global variable Routines available to access '*`errno`*' and '*`h_errno`*' variables:
  - INT PROC get_errno;
  - INT PROC get_h_errno;

These restrictions imply that the following features are not available in the C run-time library subset:

- `_MAIN`, that is, startup processing, general initialization.
- Heap management (`'malloc'`, `'calloc'`, `'realloc'`, `'free'`) is available only through the Common Run-Time Environment (CRE) user heap management routines. Refer to the *CRE Programming Manual* for details.

If mixed TAL and C code has a TAL MAIN section, the restricted set of functions just listed applies. If mixed TAL and C code having a C `_main` is used, full C run-time library functionality is available.

Usage/Bind Considerations

The following steps summarize the TAL usage and bind considerations in a CRE environment:

1. All addresses must be 32 bits (`.EXT`).
2. Source SOCKPROC to reference socket library procedures.

3. Source SOCKDEFT to reference socket library structures.
4. Specify the CRE compiler directive (`ENV COMMON`) either in the program source code or in the compilation line.

pTAL does not have access to the CRE initialization routine. For information about running a pTAL program in the CRE environment, see the *TNS/R Native Application Migration Guide*.

1. All addresses must be 32 bits (`.EXT`).
2. Source SOCKPROC to reference socket library procedures.
3. Source SOCKDEFT to reference socket library structures.
4. Specify the CRE compiler directive (`EXPORT_GLOBALS`) prior to compilation.

TAL to pTAL Conversion Issues

NOTE: For more information about socket library support for TAL and pTAL applications, see TALDOCUM in $system.ztcpip.

TAL users of the socket library converting to pTAL should use the SRL version of the socket library, ZINETSRL. For applications unable to run as a CRE compliant executable, a CRE-independent native mode socket library is provided, LNETINDN. LNETINDN is a linkable object.

The TAL-callable functions, `paramcapture()` and `allparamcapture()`, have been removed from the D40 socket library. These functions provided a mechanism to save run-time parameters used by the socket library (`=TCPIP^PROCESS^NAME`, `=TCPIP^HOSTS^FILE`, and so forth). Because the DEFINE mechanism is now utilized instead of PARAM, this functionality is no longer required.

The prototypes specified in SOCKPROC and the structure templates in SOCKDEFT have changed to conform to the native version of the socket library. Function parameter and return value data types that were specified as INT have been changed to `INT(32)`. Applications converting from TAL to pTAL must ensure that these data types are reflected accordingly in their code. Variables of type INT in existing code need to be cast to `INT(32)`, or declared as `INT(32)`, for native socket library function calls.

Defines in SOCKPROC and SOCKDEFT can be used as is with the following exception. `AF_INET` and `AF_INET6`, defined in SOCKDEFT, are declared as `INT(32)` for a pTAL compiled application. When using `AF_INET` or `AF_INET6` within the `sockaddr`, `sockaddr_in`, or `sockaddr_in6` structure, you must cast `AF_INET` or `AF_INET6` to an `INT` when assigning it to `sa_family`, `sin_family`, or `sin6_family`.

CRE Considerations

C applications using the Socket Library are compiled by the D-series C compiler. The C compiler generates code that runs in the CRE (Common Run-Time Environment). The CRE makes assumptions about the use of primary global memory, memory management, and trap handling that is incompatible with certain applications, such as the HP ODBC server. The CRE-Independent Socket Library (`LNETINDL` for the large-memory model, `LNETINDW` for the wide-data model, and `LNETINDN` for native mode) has no dependence on the CRE.

For TAL application programs that use the Socket Library, application programs must specify the ENV compiler directive `COMMON` for the D-series TAL compiler to generate code that runs in the CRE.

TAL application programs can specify the directive either in a compilation command or in the program source code before any declarations. For example, the following compilation command produces a TAL object file compiled for the CRE:

```
TAL / IN source, OUT listing /; ENV COMMON
```

NOTE: HP recommends that you use the Common Run-Time Environment (CRE) when developing TAL socket applications. CRE is described in the *CRE Programming Manual*.

If your application is incompatible with CRE, use the CRE-Independent socket library described in "Socket Libraries" at the beginning of this section.

Native Mode C/C++ Issues

Users of the native mode C/C++ compiler (nmc) need to specify the *extensions* compiler pragma for correct compilation of the socket library header files. The *extensions* pragma also needs to be specified when the c89 compiler is used for `systype=guardian` compiles.

accept

The `accept` function checks for connections on an existing waited socket. When a connection request arrives, `accept` creates a new socket to use for data transfer and accepts the connection on the new socket.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* if using IPv6 */
#include <netdb.h>

new_socket = accept (socket, from_ptr, from_len_ptr);

     int new_socket, socket;
     struct sockaddr *from_ptr;
     int *from_len_ptr;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

new_socket := accept (socket, from_ptr, from_len_ptr);

       INT(32)     new_socket;
       INT(32)     socket;
       INT .EXT    from_ptr (sockaddr_in),
           .EXT    from_len_ptr;
```

new_socket
: return value; the socket number for the new, connected socket that is created for data transfer.

  If the call is not successful, –1 is returned and the external variable *errno* is set as indicated below in Errors (page 90).

socket
: input value; specifies the socket number, created by a previous `socket` call, to be used to check for connections.

from_ptr
: input and return value; points, on return, to the remote address and port number for the new connection.

from_len_ptr
: input and return value; points, initially, to a value indicating the size in bytes of the structure pointed to by *from_ptr*. On return, it points to the actual length in bytes of the remote address and port number, or `sockaddr` data structure, pointed to by *from_ptr*.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

ECONNRESET	The connection was reset by the peer process before the accept operation completed.
EINVAL	An invalid argument was specified.

Usage Guidelines

- This is a waited call; your program is blocked until the operation completes. For nowait I/O, use accept_nw and accept_nw2.
- For TCP server applications, a call to bind and listen must precede a call to accept.
- The original socket remains in a listening state.
- Declare the *from_ptr* variable as struct sockaddr_in6 * for IPv6 use or as struct sockaddr_storage * for protocol-independent use. In C, when you make the call, cast the variable to sockaddr. (See the IPv6 example.)

Examples

The following programming example calls the accept function to accept a connection on a TCP socket.

INET: in this example, assume that fd is the socket number returned by a call to socket.

```
#include <socket.h>
#include <in.h>
#include <netdb.h>

...
struct sockaddr_in sin, from;
int flen;
char buf[256];

/* Before accept, program must call socket, bind,
 * and listen.
 */

flen = sizeof(from);
if ((s2 = accept(fd, (struct sockaddr *)&from, &flen)) < 0) {
       perror ("Server: Accept failed.");
       exit (0);
}
inet_ntop (AF_INET, &from->sin_addr, buf, INET_ADDRSTRLEN);
printf ("Server connected from remote %s, %d\n", buf, from.sin_port);
```

INET6: In this example, assume fd is the socket number returned by a call to socket.

```
#include <socket.h>
#include <in.h>
#include <in6.h>
#include <netdb.h>

...
struct sockaddr_in6, from;
int flen;
char buf[INET6_ADDRSTRLEN];

/* Before accept, program must call socket, bind,
 * and listen.
 */

flen = sizeof(from);
```

```
/* Notice from is cast to struct sockaddr in the following call
as suggested in the Usage Guidelines */
if ((s2 = accept(fd, (struct sockaddr *)&from, &flen)) < 0) {
       perror ("Server: Accept failed.");
       exit (0);
}
inet_ntop(AF_INET6, &from.sin6_addr, buf, sizeof(buf));
printf ("Server Connected from remote %s.%d\n", buf, from.sin6_port);
```

accept_nw

The `accept_nw` function checks for connections on an existing nowait socket. It is designed to be followed first by a call to `socket_nw` to create a new socket, then a call to `accept_nw2` to accept the connection on the new socket.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* if using IPv6 */
#include <netdb.h>

error = accept_nw (socket, from_ptr, from_len1, tag);

        int error, socket;
        struct sockaddr *from_ptr;

        int *from_len1;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := accept_nw (socket, from_ptr, from_len1, tag);

       INT(32)    error;
       INT(32)    socket;
       INT .EXT   from_ptr (sockaddr_in);
       INT .EXT   from_len1;
       INT(32)    tag;
```

error
: return value. If the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 201).

socket
: input and return value; specifies the socket number, created by a previous `socket_nw` call, to be used to check for connections.

from_ptr
: input and return value; points, on return, to the remote address and port number for the new connection from which the connection was initiated.

from_len1
: input and return value; points to a value indicating the size in bytes of the structure pointed to by *from_ptr*. Set the `from_len1` used in the `accept_nw` call to point to the size of the `sockaddr` struct before making the call. `accept_nw` then returns the remote client's IP address in the `from_ptr` parameter of the `sockaddr` or `sockaddr_in6` struct. This is an input parameter.

tag
input value; the *tag* parameter to be used for the nowait operation.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

EALREADY	There is already an outstanding call on the socket.
ECONNRESET	The connection was reset by the peer process before the accept_nw operation completed.
EINVAL	An invalid argument was specified.

Usage Guidelines

- The accept_nw function is designed to be followed first by a call to socket_nw to create a new socket, then by a call to accept_nw2. The *from_ptr* parameter from accept_nw is passed to accept_nw2, which accepts the connection on the new socket.
- Use accept_nw2 after this call.
- This is a nowait call; it must be completed with a call to the Guardian procedure AWAITIOX. For a waited call, use accept.
- The accept_nw call causes TCP/IP to do a listen and accept in one call.
- Declare the *from_ptr* variable as type struct sockaddr_in6 * for IPv6 use or as type struct sockaddr_storage * for protocol-independent use. In C, when you make the call, cast the variable to sockaddr *. (See the IPv6 example.)
- For the Conventional TCP/IP product only, it is not recommended to use CANCEL or CANCELREQ calls with accept_nw. While this procedure works as expected with IPv6 and CIP products, with Conventional TCP/IP, accept_nw causes a pending incoming connection to be immediately marked as allocated and the cancel does not undo this. Subsequent accept_nw requests require additional incoming connections for the requests to complete. Because the connection has been marked allocated it cannot be accepted by a subsequent accept_nw.

See Nowait Call Errors (page 86) for information on error checking.

See Chapter 3 (page 62) for information about struct sockaddr, struct sockaddr_in6, and struct sockaddr_storage.

Example

INET: The following IPv4 TCP server programming example calls accept_nw, socket_nw, and accept_nw2. This call accepts a connection on the new socket fd2 created for nowait data transfer.

```
#include <socket.h>
#include <in.h>
#include <netdb.h>
...
struct sockaddr_in from;

...
if ((fd1 = socket_nw(AF_INET, SOCK_STREAM,,1,1)) < 0) {
       perror ("Server Socket 1 create failed.");
       exit (0);
    /* Call AWAITIOX */

}
/* Before calling accept_nw, program must call bind_nw and
 * listen. A call to AWAITIOX must follow the bind_nw call.
 */
```

```
flen = sizeof(from);
if ((cc = accept_nw(fd1, (struct sockaddr *)&from, flen,
             t)) < 0) {
       perror ("Server: Accept failed.");
       exit (0);
}
else {
    /* Call AWAITIOX using socket fd1 and tag t. */
...
}

if ((fd2 = socket_nw(AF_INET, SOCK_STREAM,,1,1)) < 0) {
       perror ("Server Socket 2 create failed.");
       exit (0);
}
else {
    /* Call AWAITIOX using socket fd2. */
}...
if ((cc = accept_nw2(fd2, (struct sockaddr *)&from, t)) < 0) {
       perror ("Server: Accept failed.");
       exit (0);
}
else {
    /* Call AWAITIOX using socket fd2 and tag t. */
...
}
```

INET6: the following Parallel Library TCP/IP IPv6 server programming example calls `accept_nw`, `socket_nw`, and `accept_nw2`. This call accepts a connection on the new socket `fd2` created for nowait data transfer.

```
#include <socket.h>
#include <in.h>
#include <in6.h>
#include <netdb.h>
...
struct sockaddr_in6 from;

...
if ((fd1 = socket_nw(AF_INET6, SOCK_STREAM,,1,1)) < 0) {
       perror ("Server Socket 1 create failed.");
       exit (0);
    /* Call AWAITIOX */

}
/* Before calling accept_nw, program must call bind_nw and
 * listen. A call to AWAITIOX must follow the bind_nw call.
 */

flen = sizeof(from);
/* Notice that from is cast as struct sockaddr * as suggested in
the Usage Guidelines */
if ((cc = accept_nw(fd1, (struct sockaddr *)&from, flen,
             t)) < 0) {
       perror ("Server: Accept failed.");
       exit (0);
}
else {
      /* Call AWAITIOX using socket fd1 and tag t. */
...

}
   if ((fd2 = socket_nw(AF_INET6, SOCK_STREAM,,1,1)) < 0) {
```

```
        perror ("Server Socket 2 create failed.");
        exit (0);
}
else {
    /* Call AWAITIOX using socket fd2. */
}...
if ((cc = accept_nw2(fd2, (struct sockaddr *)&from, t)) < 0) {
        perror ("Server: Accept failed.");
        exit (0);
}
else {
      /* Call AWAITIOX using socket fd2 and tag t. */
...
}
```

accept_nw1

accept_nw1 can be used instead of accept_nw; use accept_nw1 to set the maximum connections in the queue awaiting acceptance on a socket.

C Synopsis

```
#include <socket.h>

#include <in.h.>
#include <in6.h> /* if using IPv6 */
#include <netdb.h>

error = accept_nw1 (socket, from_ptr, from_len1, tag, queue_length);

        int error, socket;
        struct sockaddr *from_ptr;
        int *from_len1;
        long tag;
        int queue_length;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := accept_nw1 (socket, from_ptr, from_len1, tag, queue_length);

        INT(32)  error;
        INT(32)  socket;
        INT .EXT from_ptr (sockaddr_in);
        INT.EXT  from_len1;
        INT(32)  tag;
        INT(32)  queue_length;
```

error
return value; if the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 95).

socket
input value; specifies the socket number, created by a previous socket_nw call, to be used to check for connections.

from_ptr
input and return value; points, on return, to the remote address and port number for the new connection from which the connection was initiated.

from_len1
input and return value; points to a value indicating the size in bytes of the structure pointed to by *from_ptr*.

tag
input value; the *tag* parameter to be used for the nowait operation.

queue_length
input value; specifies the maximum queue length (number of pending connections). Values are 1 through 128.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

EALREADY	There is already an outstanding call on the socket.
ECONNRESET	The connection was reset by the peer process before the accept_nw operation completed.
EINVAL	An invalid argument was specified.

Usage Guidelines

- Use the accept_nw1 call instead of the accept_nw call when you need to set the queue length.
- This is a nowait call; it must be completed with a call to the Guardian procedure AWAITIOX. For a waited call, use accept.
- The accept_nw1 call causes TCP/IP to do a listen and accept in one call.
- The accept_nw1 function must be followed first by a call to socket_nw to create a new socket and then by a call to accept_nw2. The *from_ptr* parameter from accept_nw1 is passed to accept_nw2, which accepts the connection on the new socket.
- Use accept_nw2 after this call.
- Declare the *from_ptr* variable as struct sockaddr_in * for IPv6 use or as struct sockaddr_storage * for protocol-independent use. In C, when you make the call, cast the variable to sockaddr *.
- For the Conventional TCP/IP product only, it is not recommended to use CANCEL or CANCELREQ calls with accept_nw1. While this procedure works as expected with IPv6 and CIP products, with Conventional TCP/IP, accept_nw1 causes a pending incoming connection to be immediately marked as allocated and the cancel does not undo this. Subsequent accept_nw1 requests require additional incoming connections for the requests to complete. Because the connection has been marked allocated it cannot be accepted by a subsequent accept_nw1.

accept_nw2

The accept_nw2 function accepts a connection on a new socket created for nowait data transfer. Before calling this procedure, a program should call accept_nw on an existing socket and then call socket_nw to create the new socket to be used by accept_nw2.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* For IPv6 use */
#include <netdb.h>

error = accept_nw2 (new_socket, from_ptr, tag);

        int error, new_socket;
```

```
        struct sockaddr *from_ptr;

        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := accept_nw2 (new_socket, from_ptr, tag);

        INT(32)    error;
        INT(32)  new_socket;
        INT   .EXT from_ptr(sockaddr_in);
        INT(32)    tag;
```

error
: return value; f the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 96).

new_socket
: input value; the socket number for the new socket on which the connection is to be accepted, as returned by a call to socket_nw.

from_ptr
: input value; points to the address and port number returned from the call to accept_nw or accept_nw1.

tag
: input value; the *tag* parameter to be used for the nowait operation.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

EADDRINUSE	accept_nw2() posted on an already-bound socket. (For Parallel Library TCP/IP and NonStop TCP/IPv6 only.)
EALREADY	Operation is already in progress. (For Parallel Library TCP/IP and NonStop TCP/IPv6 only.)
ECONNRESET	The connection was reset by the peer process before the accept operation completed. This error also can be received when a call was done on a socket when the socket was in an incorrect state.
EINVAL	An invalid argument was specified.
EISCONN	Socket is already connected. (For Parallel Library TCP/IP and NonStop TCP/IPv6 only.)
ENOBUF	No Buffer Space available. (For Parallel Library TCP/IP and NonStop TCP/IPv6 only.)
ERSCH	The socket specified in the new_socket parameter was invalid. Close the socket using the FILE_CLOSE call. Repeat the accept_nw, socket_nw and accept_nw2 sequence of calls.

Usage Guidelines

- This is a nowait call; it must be completed with a call to the AWAITIOX procedure. For a waited call, use accept.
- The accept_nw and accept_nw2 functions work together. The accept_nw function checks for connections on an existing nowait socket. When a connection request arrives, it returns the address and port number from which the connection request came. A new socket is then created with socket_nw. Finally, the new socket number returned by socket_nw and the

address-port number combination returned by accept_nw is passed to accept_nw2 to establish the connection on the new socket.

- The call to accept_nw made prior to this call may be made in another process, such as the LISTNER process.
- Declare the *from_ptr* variable as struct sockaddr_in6 * for IPv6 use or as struct sockaddr_storage * for protocol-independent use. In C, when you make the call, cast the variable to sockaddr *. (See the IPv6 example.)
- Applications doing ACCEPT_NW2 calls can only see listening applications in the same LNP. (H-series and G06.22 and later G-series RVUs of NonStop TCP/IPv6 only.)

Example

See accept_nw (page 91), which also calls accept_nw2.

accept_nw3

The accept_nw3 function accepts a connection on a new socket created for nowait data transfer. Before calling this procedure, a program should call accept_nw on an existing socket and then call socket_nw to create the new socket to be used by accept_nw3.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* For IPv6 use */
#include <netdb.h>

error = accept_nw3 (new_socket, from_ptr, me_ptr, tag);

        int error, new_socket;
        struct sockaddr *from_ptr, *me_ptr;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := accept_nw3 (new_socket, from_ptr, me_ptr, tag);

        INT(32)   error;
        INT(32)  new_socket;
        INT  .EXT from_ptr(sockaddr_in);
        INT  .EXT me_ptr(sockaddr_in);
        INT(32)   tag;
```

error
return value; if the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 98).

new_socket
input value; the socket number for the new socket on which the connection is to be accepted, as returned by a call to socket_nw.

from_ptr
input value; points to the address and port number returned from the call to accept_nw or accept_nw1.

me_ptr
input value; points to the local address and port number used by bind_nw.

tag
input value; the *tag* parameter to be used for the nowait operation.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

EADDRINUSE	Accept_nw3() posted on an already-bound socket. (For Parallel Library TCP/IP and NonStop TCP/IPv6 only.)
EALREADY	Operation is already in progress. (For Parallel Library TCP/IP and NonStop TCP/IPv6 only.)
ECONNRESET	The connection was reset by the peer process before the accept operation completed. This error also can be received when a call was done on a socket when the socket was in an incorrect state.
EINVAL	An invalid argument was specified.
EISCONN	Socket is already connected. (For Parallel Library TCP/IP and NonStop TCP/IPv6 only.)
ENOBUF	No Buffer Space available. (For Parallel Library TCP/IP and NonStop TCP/IPv6 only.)
ERSCH	The socket specified in the new_socket parameter was invalid. Close the socket using the FILE_CLOSE call. Repeat the accept_nw, socket_nw and accept_nw3 sequence of calls.

Usage Guidelines

- This is a nowait call; it must be completed with a call to the AWAITIOX procedure. For a waited call, use accept.
- The accept_nw and accept_nw3 functions work together. The accept_nw function checks for connections on an existing nowait socket. When a connection request arrives, the accept_nw function returns the address and port number from which the connection request came. A new socket is then created with socket_nw. Finally, the new socket number returned by socket_nw and the address-port number combination returned by accept_nw is passed to accept_nw3 to establish the connection on the new socket.
- The call to accept_nw made prior to this call can be made in another process, such as the LISTNER process.
- Declare the *from_ptr* and *me_ptr* variables as struct sockaddr_in6 * for IPv6 use or as struct sockaddr_storage * for protocol-independent use. In C, when you make the call, cast the variable to sockaddr *. (See the IPv6 example.)
- Applications doing ACCEPT_NW3 calls can only see listening applications in the same LNP. (H-series and G06.22 and later G-series RVUs of NonStop TCP/IPv6 only.)

bind, bind_nw

The bind and bind_nw functions associate a socket with a specific local Internet address and port number.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

error = bind (socket, address_ptr, address_len);

error = bind_nw (socket, address_ptr, address_len, tag);
```

```
        int error, socket;

        struct sockaddr *address_ptr;
        int address_len;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := bind (socket, address_ptr, address_len);

error := bind_nw (socket, address_ptr, address_len, tag);

        INT(32)   error, socket;

        INT .EXT address_ptr(sockaddr_in);
        INT(32)  address_len;
        INT(32)  tag;
```

error
: return value; if the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 99).

socket
: input value; specifies the socket number for the socket, as returned by the call to `socket` or `socket_nw`.

address_ptr
: input value; points to the address-port number combination based in the structure `sockaddr`, `sockaddr_in6`, or `sockaddr_storage`, to which the socket is to be bound.

  If the address in the `sin_addr` field of the structure is `INADDR_ANY`, connections are accepted from hosts on any network. If the port number in the `sin_port` field of the structure is zero, the next available port is assigned. Port numbers 0 to 1023 are reserved for use by predefined services, such as TELNET. If the port number is in the range 0 through 1023 (known as reserved ports), the process access ID of the requesting application must be in the SUPER group (user ID 255,*nnn*).

  For NonStop TCP/IPv6, if the address in the `sin6_addr` field of the structure is `in6addr_any`, connections are accepted from hosts on any network. If the port number in the `sin6_port` field of the structure is zero, the next available port is assigned. Port numbers 0 to 1023 are reserved for use by predefined services, such as TELNET. If the port number is in the range 0 through 1023 (known as reserved ports), the process access ID of the requesting application must be in the SUPER group (user ID 255,*nnn*).

address_len
: input value; *address_len* is maintained only for compatibility and should be a value indicating the size in bytes of the structure (the remote address and port number) pointed to by *address_ptr*.

tag
: input value; the *tag* parameter to be used for the nowait operation initiated by `bind_nw`.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

`EADDRNOTAVAIL`	The specified IP address and port number was not available on the local host.
`EADDRINUSE`	The specified IP address and port number was already in use.

EINVAL	The specified socket was already bound to an address, or the address_len was incorrect.
EACCES	The specified address cannot be assigned to a nonprivileged user.

Usage Guidelines

- Use `bind` on a socket created for waited operations, or `bind_nw` on a socket created for nowait operations. The operation initiated by `bind_nw` must be completed with a call to the `AWAITIOX` procedure.

  NOTE: The socket goes into a TCP LISTEN state after the application completes a bind on an IP address and port. There is a possibility that TCP/IP can receive a connection on that socket if a rogue client tries to connect to that IP address and port.

- Multiple sockets created by different processes can be bound to the same UDP port. When a broadcast message arrives on the UDP port, only one process is notified. TCP/IP determines which process to notify based on the network address portion of the Internet address. If the network address of a socket is the same as the network address of the broadcast message, the process that created and bound the socket is notified. For example, assume these sockets A, B, and C are created by different processes and are bound to port 67 using the following Internet addresses:

  ```
  Socket A   130.252.12.8
  Socket B   130.252.10.8
  Socket C   10.5.0.9
  ```

  A UDP broadcast message addressed to 130.252.10.255 (port 67) arrives on the socket bound to port 67 with Internet address 130.252.10.8. The process that created socket B is notified because the network address of the socket matches the network address of the broadcast message. (In the Berkeley sockets interface, the socket most recently bound to the port is notified.)

  Only one socket can be bound to a particular combination of Internet address and port number.

- **UDP Port Considerations for Parallel Library TCP/IP and NonStop TCP/IPv6**. If a process maintains a context for its messages, the process should not use round robin on shared UDP ports.

  The processes sharing the UDP port should not maintain a context for previous messages because there is no guarantee that a sequence of messages is delivered to the same socket if the port is shared. In fact, with round-robin enabled, a sequence of messages is distributed to each of the port-sharing sockets, in turn.

  For example, TFTP server assumes that all packets from a given source are directed to it (the TFTP server process). This assumption about the source is what is meant by maintaining a "context." Because TFTP server makes that assumption about packets from a given source, that is, maintains that "message context," it must be the only TFTP server process on that UDP port. If another TFTP server is sharing the UDP port, packets from the same source could go to two different TFTP server processes resulting in one of the TFTP servers missing some of the packets destined for it.

  For applications that must maintain a context across multiple messages received (such as TFTP server and WANBOOT), if you want multiple instances running in parallel, you can circumvent the problem introduced by round robin by changing the application to bind to the SUBNET IP address rather than to INADDR_ANY or IN6ADDR_ANY. Binding to the IP address allows one instance of the application for each SUBNET to be supported by Parallel Library TCP/IP and NonStop TCP/IPv6 with sharing of the same port number. NonStop TCP/IPv6 and NonStop TCP/IP then distributes incoming packets that came in from one SUBNET only to the application that bound to that SUBNET. This circumvents the problem introduced by round-robin distribution of incoming packets among sockets sharing the same port.

Alternatively, for NonStop TCP/IP, you can use LNP to create multiple TCP6SAM processes, each with its own IP address, similar to the multiple- TCP/IP process technique of conventional TCP/IP. (See Multiple NonStop TCP/IP Processes and Logical Network Partitioning (LNP) (NonStop TCP/IPv6, H-Series and G06.22 and Later G-Series RVUs Only) (page 43).)

- **TCP Port Considerations for NonStop TCP/IP and NonStop TCP/IP.** If you have used the following DEFINE to set up round-robin filtering for Parallel Library TCP/IP or NonStop TCP/IPv6, consider the following for socket programming.

  ```
  ADD DEFINE =PTCPIP^FILTER^KEY, class map, file file-name
  ```

  Round-robin filtering allows multiple binds to the same IP and port if more than one application per processor is binding to the port at one time. Furthermore, the multiple binds to the same IP port can only come from processes that share the same NonStop TCP/IP or Parallel Library TCP/IP filter key (the variable *file name* in the DEFINE).

  You can limit the shared ports by adding one or both of the following defines:

  ```
  ADD DEFINE =PTCPIP^FILTER^TCP^PORTS, FILE P/Pendport

  ADD DEFINE =PTCPIP^FILTER^UDP^PORTS, FILE Pstartport.Pendport
  ```

  The *startport* and *endport* variables are integers specifying the allowable port range. The `=PTCPIP^FILTER^TCP^PORTS` key limits the shared TCP ports to the range defined in *startport* and *endport*. The `=PTCPIP^FILTER^UDP^PORTS` key limits the shared UDP ports to the range defined in *startport* and *endport*. Ports outside those ranges are not shared.

  See the *TCP/IPv6 Configuration and Management Manual* for more information about DEFINE.

- See Nowait Call Errors (page 86) for information on error checking.
- Declare the *address_ptr* variable as `struct sockaddr_in6 *` for IPv6 use or as `struct sockaddr_storage *` for protocol-independent use. In C, when you make the call, cast the variable to `sockaddr`. (See the IPv6 example.)

Examples

INET: the following IPv4 programming example calls the `bind` routine. The socket `fd` is bound to the address and port number in the `sin` structure:

```
#include <socket.h>
#include <in.h>
#include <netdb.h>
...
struct sockaddr_in sin;
...
/* The code here (not shown) should create a socket fd.
 * Then the local address and port number
 * in the sin structure are set up. The port number is passed
 * as an argument when the program is run.
*/
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = port;
if (bind (fd, (struct sockaddr *)&sin, sizeof (sin)) < 0) {
        perror ("SERVER: Bind failed.");
        exit (1);
}
/* Bind call succeeded. /*
```

INET6: the following IPv6 programming example calls the `bind` routine. The socket `fd` is bound to the address and port number in the `sin` structure:

```
#include <socket.h>
#include <in.h>
#include <in6.h>
```

```
#include <netdb.h>
...
struct sockaddr_in6 sin;
...
/* The code here (not shown) should create a socket fd.
 * Then the local address and port number
 * in the sin structure are set up. The port number is passed
 * as an argument when the program is run.
*/
sin.sin6_family = AF_INET6;
sin.sin6_addr = in6addr_any;
sin.sin6_port = port;

/* Notice that sin is cast as sockaddr as suggested in the Usage Guidelines */
if (bind (fd, (struct sockaddr *)&sin, sizeof (sin)) < 0) {
        perror ("SERVER: Bind failed.");
        exit (1);
}
/* Bind call succeeded. /*
```

connect, connect_nw

The `connect` and `connect_nw` functions connect the specified socket to a remote socket.

For TCP, these functions request an active connection. For UDP or IP, they permanently specify the destination address for the socket.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

error = connect (socket, address_ptr, address_len);

error = connect_nw (socket, address_ptr, address_len, tag);

        int error, socket;
        struct sockaddr *address_ptr;

        int address_len;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := connect (socket, address_ptr, address_len);

error := connect_nw (socket, address_ptr, address_len, tag);

        INT(32)    error;
        INT(32)    socket;
        INT  .EXT address_ptr (sockaddr_in);
        INT(32)    address_len;
        INT(32)    tag;
```

error

return value; if the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 103). Refer to Appendix B (page 243), for more details.

socket
input value; specifies the socket number for the socket, as returned by the call to `socket` or `socket_nw`.

address_ptr
input value; points to the address and port number (based on the structure `sockaddr_in`, `sockaddr_in6`, `sockaddr_storage`) of the remote socket to which a connection is to be established.

address_len
input value; should be a value indicating the size in bytes of the remote address and port number pointed to by *address_ptr*.

tag
input value; the *tag* parameter to be used for the nowait operation initiated by `connect_nw`.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

`EALREADY`	There is already an outstanding call on the socket.
`EISCONN`	The specified socket was already connected.
`ETIMEDOUT`	The connection timed out without being established.
`ECONNREFUSED`	The remote host rejected the connection.
`ENETUNREACH`	The network (of the remote host) was unreachable.
`EINVAL`	One of the arguments to this call was invalid.

Usage Guidelines

- Use `connect` on a socket created for waited operations, or `connect_nw` on a socket created for nowait operations. The operation initiated by `connect_nw` must be completed with a call to the `AWAITIOX` procedure.
- For more information on checking errors, see Nowait Call Errors (page 86).
- For more information about `struct sockaddr *`, `struct sockaddr_in6` and `sockaddr_storage`, see Chapter 3 (page 62). Also, see getaddrinfo (page 107) and addrinfo (page 64).
- Declare the *address_ptr* variable as `struct sockaddr_in6 *` for IPv6 use or as `struct sockaddr_storage *` for protocol-independent use. In C, when you make the call, cast the variable to `sockaddr *`. (See the "Examples" (page 103).)

NOTE: Using CIP, when trying to connect to a remote IPv6 link-local address might fail with error `EINVAL`. This error is displayed when:

- The socket is not bound to the IPv6 link-local address on the local interface, or
- The scope ID (`sin6_scope_id` member in `struct sockaddr_in6`) is not specified.

Examples

INET: The following programming example calls the `connect` routine that connects the socket `fd` to a remote socket. The remote structure contains the address and port of the remote socket:

```
#include <socket.h>
#include <in.h>
#include <netdb.h>
...
struct sockaddr_in remote;
....
```

```
/* Program must contain code to create the socket fd
 * and to fill in the remote address before calling connect.
 */
...
if (connect (fd,(struct sockaddr *)&remote,sizeof(remote))  <0) {
        perror ("Client failed to connect to remote host.");
        exit (0);
}
printf ("CLIENT:Connected ...\n");
```

INET6: The following programming example calls the `connect` routine that connects the socket `fd` to a remote socket. The remote structure contains the address and port of the remote socket:

```
#include <socket.h>
#include <in.h>
#include <in6.h>
#include <netdb.h>
...
struct sockaddr_in6 remote;
....
/* Program must contain code to create the socket fd
 * and to fill in the remote address before calling connect.
 */
...
/*Notice that remote is cast as struct sockaddr as suggested in
the Usage Guidelines */
if (connect (fd,(struct sockaddr *)&
        remote,sizeof(remote))  <0) {
        perror ("Client failed to connect to remote host.");
        exit (0);
}
printf ("CLIENT:Connected ...\n");
```

freeaddrinfo

The `freeaddrinfo` function frees the memory of one or more `addrinfo` structures previously created by the `getaddrinfo` function. Any dynamic storage pointed to by the structure is also freed. (This function is supported for NonStop TCP/IPv6 only.)

C Synopsis

```
#include <netdb.h>

void freeaddrinfo (struct addrinfo *ai);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

  freeaddrinfo (ai);

  INT .EXT ai (addrinfo);
```

ai
input value; specifies the `addrinfo` structure to be freed.

Errors

No errors are returned for this function.

Usage Guidelines

Call this function once for each structure created by calls to `getaddrinfo` before closing a socket. Upon successful completion, `freeaddrinfo` does not return a value. The address information structure and associated storage are returned to the system.

Examples

INET6: the following IPv6 programming example calls the `freeaddrinfo` routine after the `getaddrinfo` function returns a value:

```
#include <netdb.h>
...
struct addrinfo *res;
struct addrinfo *aip;
for(aip = res; aip!= NULL; aip = aip->ai_next){
/*create a socket, address type depends on getaddrinfo()
returned value */
    sock=socket(aip->ai_family, aip->ai_socktype,
      aip->ai_protocol);
     if (sock == -1) {
        perror("socket");
        freeaddrinfo(res);
          return(-1);
   }
}
```

freehostent

The `freehostent` function frees the memory of one or more `hostent` structures returned by the `getipnodebyaddr` or `getipnodebyname` functions. (This function is supported for NonStop TCP/IP only.)

C Synopsis

```
#include <netdb.h>

void freehostent(struct hostent *ptr);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

    freehostent(ptr);

    INT .EXT ptr(hostent);
```

ptr
input value; a pointer to the structure `hostent`.

Usage Guidelines

Call this function once for each `hostent` structure returned by the `getipnodebyaddr` or `getipnodebyname` functions.

gai_strerror

The `gai_strerror` function aids applications in printing error messages based on the EAI_xxx codes returned by the getaddrinfo function. The IPv6 functions `getipnodebyaddr`, `getipnodebyname`, `getaddrinfo`, and `getnameinfo` return errors in a thread-safe structure.

The gai_strerror function call returns a pointer to a character string describing the error code passed into it. (This function is supported for Parallel Library TCP/IP only.)

C Synopsis

```
#include <netdb.h>

char *gai_strerror (int ecode);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

return_value := gai_strerror (ecode);

    INT(32) return_value;
    INT(32) ecode;
```

return_value
is a pointer to a string described in ecode.

ecode
input value; specifies one of the following error codes returned by the getaddrinfo function; the returned strings are as follows:

Error Codes and Returned Strings	Reason
EAI_ADDRFAMILY: See the EAI_FAMILY returned string. EAI_ADDRFAMILY is defined but never returned.	Address family for *hostname* not supported.
EAI_AGAIN:"The name could not be resolved this time. Future attempts may succeed."	Temporary failure in name resolution.
EAI_BADFLAGS:"The flags parameter has an invalid value."	Invalid value for *ai_flags*.
EAI_FAIL:"A non-recoverable error occurred."	Non-recoverable error in name resolution.
EAI_FAMILY:"Address family was not recognized or address length was invalid."	*ai_family* not supported.
EAI_MEMORY:"Memory allocation failure."	Memory allocation failure.
EAI_NONAME:"Name does not resolve to supplied parameters."	Neither *hostname* nor *servname* supplied or the name does not resolve using the supplied parameters.
EAI_SERVICE:"The service passed was not recognized for the specified socket type."	*servname* not supported for *ai_socktype*.
EAI_SOCKTYPE:"The intended socket type was not recognized."	*ai_socktype* not supported.
EAI_SYSTEM:"A system error occurred; error code found in errno."	System error returned in *errno*.

Usage Guidelines

Call this function to aid in printing human-readable error messages based on the EAI_xxx error codes returned by the getaddrinfo function.

Example

The following programming example calls the gai_strerror routine to print error messages:

```
error = getaddrinfo(hostname, servicename, &hints, &res);
if(error != 0) {
  (void)fprintf(stderr,"myFunction: getaddrinfo returned error
   %i ", error);
  (void)fprintf(stderr,"%s0", gai_strerror(error));
return -1;
}
```

Errors

errno is set only on the return of EAI_SYSTEM. See *ecode* for further information about error codes.

getaddrinfo

The getaddrinfo function converts hostnames and service names into socket address structures. (This function is supported for NonStop TCP/IPv6 only.)

NOTE: The C synopsis is given in ANSI C format rather than the pre-ANSI C formats of the other library routines because the only NonStop servers you can use these routines on all support ANSI C. (ANSI C format defines the function and the arguments in the same line rather than using an assign statement and defining the arguments underneath.)

C Synopsis

```
#include <netdb.h>
int getaddrinfo (const char *hostname, const char *service,
const struct addrinfo *hints, struct addrinfo **result);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

    error := getaddrinfo (hostname, service, hints, result);

    INT(32)      error;
    STRING .EXT hostname;
    STRING .EXT service;
    INT .EXT    hints(addrinfo);
    INT .EXT    result(addrinfo);
```

error

return value; it is 0 upon success or a nonzero error code upon failure. The error codes are described in gai_strerror (page 105).

hostname

input value; specifies a pointer to a character representing one of the following:

- An Internet node hostname.
- An IPv4 address in dotted-decimal format.
- An IPv6 address in hexadecimal format.
- NULL if no hostname requires converting; when NULL is used, either *service* or *hints* must be non-NULL.

service
input value; specifies a pointer to a character representing one of the following:

- A network service name.
- A decimal port number.
- NULL if no service name requires converting; when NULL is used, either hostname or hints must be non-NULL.

hints
input value; specifies one of the following:

- A pointer to an addrinfo struct for a socket; the format of the addrinfo structure is defined in the header file netdb.h.
- NULL if no struct is available; when NULL is used, either *hostname* or *service* must be non-NULL.

result
return value; points to a list of addrinfo structs upon successful completion. (See Usage Guidelines (page 108).)

Example

This fragment of an IPv6 TCP Client shows a client that requests a service called example.

```
struct addrinfo *res, *ainfo;
struct addrinfo hints;
/* clear out hints */
memset ((char *)&hints, 0, sizeof(hints));
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo(argv[1], "example", &hints, &res);
if (error != 0) {
     fprintf(stderr, "%s: %s not found in name service database\n",
     argv[0], argv[1]);
     exit(1);
}
for (ainfo = res; ainfo != NULL; ainfo = ainfo->ai_next) {
     /* Create the socket. */
     s = socket (ainfo->ai_family,ainfo->ai_socktype,
          ainfo->ai_protocol);
     if (s == -1) {
          perror(argv[0]);
          fprintf(stderr, "%s: unable to create socket\n", argv[0]);
          freeaddrinfo(res);
          exit(1);
     }
    if (connect(s, ainfo->ai_addr, ainfo->ai_addrlen) == -1){
          perror(argv[0]);
          fprintf(stderr, "%s: unable to connect to remote\n", argv[0]);
          FILE_CLOSE(S);
          continue;
    }
     else
          break;
}
```

Usage Guidelines

- This function, along with getipnodebyname (page 116), are protocol-independent replacements for gethostbyname, host_file_gethostbyname (page 110). getaddrinfo provides extra

functionality beyond what getipnodebyname provides because getaddrinfo handles both the hostname and the service.

- Appropriate use of this function can eliminate calls to getservbyname and at the same time provide protocol independence.
- getaddrinfo function converts hostnames and service names into socket address structures. You allocate a hints structure, initialize it to 0 (zero), fill in the needed fields, and then call this function.
- This function returns through the *result* pointer a linked list of addrinfo structs that you can use with other socket functions. For a description of the addrinfo struct, see addrinfo (page 64). Each addrinfo struct contains the following members:
- A TCP or UDP client typically specifies non-NULL values for both the hostname and service parameters. A TCP client loops through all the returned socket address structures, calling the socket and connect functions for each address until a connection succeeds. A UDP client calls connect or the sendto function.
- A TCP or UDP server typically specifies a non-NULL value for service but not hostname. It also specifies the AI_PASSIVE flag in the hints struct. The returned socket address structs should contain the IP address INADDR_ANY or in6addr_any. A TCP server then calls the socket, bind, and listen functions. A UDP server calls the socket, bind, and the recvfrom functions.
- If the client or server handles only one type of socket, set hints.ai_socktype to SOCK_STREAM or SOCK_DGRAM to avoid having multiple addrinfo structs returned.
- Upon successful completion, this function returns 0 (zero), and *result* points to a new address information structure. Otherwise, getaddrinfo returns the error codes described in *ecode* .
- The freeaddrinfo (page 104) function returns the storage allocated by the getaddrinfo function.
- Ensure that the protocol file ($SYSTEM.ZTCPIP.PROTOCOL on the Guardian side or /etc/protocols on the OSS side) exists. This helps to avoid the following error:

  ENOENT(4002): No such file or directory.

gethostbyaddr, host_file_gethostbyaddr

The gethostbyaddr and host_file_gethostbyaddr functions get the name of the host with the specified Internet address. These functions are for INET addresses only; for protocol-independent applications, see getnameinfo (page 117) or getipnodebyaddr (page 114). Although these two functions provide the same service, they accomplish the service in different ways. To determine which function best suits your purpose, see the Usage Guidelines (page 110).

C Synopsis

```
#include <socket.h>
#include <netdb.h>

host_entry_ptr = gethostbyaddr (host_addr_ptr, length,
                                addr_type);

host_entry_ptr = host_file_gethostbyaddr (host_addr_ptr,
                                          length, addr_type);

      struct hostent *host_entry_ptr;
      char *host_addr_ptr;
      int length, addr_type;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

host_entry_ptr := gethostbyaddr (host_addr_ptr, length,
                                 addr_type);

host_entry_ptr := host_file_gethostbyaddr (host_addr_ptr,
                                           length, addr_type);

      INT(32)     host_entry_ptr;
      STRING .EXT host_addr_ptr;
      INT(32)     length,
                  addr_type;
```

host_entry_ptr
return value; points to a structure (based on the hostent structure) in which information on the specified host is returned. The information includes the official name, aliases, and addresses for the host.

If the lookup fails, NULL is returned and the external variable *h_*errno is set as indicated in Errors (page 110).

host_addr_ptr
input value; points to the Internet address of the host whose name is to be found. The address pointed to is in binary format and network order. (This address is in the same format and order as the return value of the function inet_addr (page 134).)

length
input value; the length of the Internet address pointed to by *host_addr_ptr*.

addr_type
input value; the type of address specified. Its value must be AF_INET.

Errors

If an error occurs, the external variable *h_*errno is set to one of the following values:

HOST_NOT_FOUND	The specified host was not found. This is the only possible value if the resolver code has been disabled.
TRY_AGAIN	The local server did not receive a response from an authoritative server. Try again later.
NO_RECOVERY	An error has occurred from which there is no recovery.

Usage Guidelines

The address that is returned in *host_entry_ptr* can be used directly in a sockaddr_in structure. The address is in network order.

The gethostbyaddr and host_file_gethostbyaddr library routines are for INET use only. For IPv6, use the getnameinfo or library routines (see getnameinfo (page 117)).

gethostbyname, host_file_gethostbyname

The gethostbyname and host_file_gethostbyname functions get the Internet address of the host whose name is specified. These functions are for INET applications only; for protocol-independent applications, see getaddrinfo (page 107) or getipnodebyname (page 116). Although these two functions provide the same service, they accomplish the service in different ways. To determine which function best suits your purpose, see the Usage Guidelines (page 111).

C Synopsis

```
#include <socket.h>
#include <netdb.h>
```

```
host_entry_ptr = gethostbyname (host_name_ptr);

host_entry_ptr = host_file_gethostbyname (host_name_ptr);

      struct hostent *host_entry_ptr;
      char *host_name_ptr;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

host_entry_ptr := gethostbyname (host_name_ptr);

host_entry_ptr := host_file_gethostbyname (host_name_ptr);

      INT(32)      host_entry_ptr;
      STRING  .EXT host_name_ptr;
```

host_entry_ptr

return value; points to a structure (based on the hostent structure) in which information on the specified host is returned. The information includes the official name, aliases, and addresses for the host.

If the lookup fails, NULL is returned, and the external variable *h_*errno is set as indicated in Errors (page 111).

host_name_ptr

input value; points to either the official name or an alias of the host whose Internet address is to be found.

Errors

If an error occurs, the external variable *h_*errno is set to one of the following values:

HOST_NOT_FOUND	The specified host was not found. This is the only possible value if the resolver code has been disabled.
TRY_AGAIN	The local server did not receive a response from an authoritative server. Try again later.
NO_RECOVERY	An error has occurred from which there is no recovery.
NO_ADDRESS	The specified hostname is valid, but the host does not have an IP address.

Usage Guidelines

- The `gethostbyname()` function is used for resolving names with hosts file. You can choose host file, external dns server, or a combination of host file and external dns server to resolve the host name.
- The parameters passed to the `gethostbyname` and `host_file_gethostbyname` functions are case-sensitive.
- The `hostent` structure is statically declared. Subsequent calls to `gethostbyname` or `host_file_gethostbyname` replace the existing data in the *hostent* structure.

NOTE: The function `host_file_gethostbyname()` supports only local hosts file.

Example

The address pointed to by *host_entry_ptr*, which is already in network order, can be used directly in a `sockaddr_in` structure, as in the following example:

```
struct sockaddr_in sin;
struct hostent *hp;
 ...
if ((hp = gethostbyname (nameptr)) != (struct hostent *)
      NULL) {

    memmove ((char *)&sin.sin_addr.s_addr,
             (char *)hp -> h_addr,
             (size_t) hp -> h_length );
  } ...
```

If the return value is not NULL, the pointer `hp` is used to move the address from the `h_addr` field of the `hp` structure to the Internet address field of the `sin` structure.

gethostbyname2

The `gethostbyname2` function gets the Internet address (IPv4 or IPv6) of the host whose name is specified. `gethostbyname2` works like `gethostbyname` but also allows specifying the address family to which the returned Internet address must belong. (This function is supported for G06.27 and later G-series RVUs and H06.05 and later H-series RVUs of NonStop TCP/IPv6.)

C Synopsis

```
#include <netdb.h>

host_entry_ptr = gethostbyname2(name, af);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

host_entry_ptr := gethostbyname2(name, af);
    INT(32) host_entry_ptr;
    STRING .EXT name;
    INT af;
```

`host_entry_ptr`
: return value; points to a structure (based on the `hostent` structure) in which information on the specified host is returned. The information includes the official name, aliases, and addresses for the host. If the lookup fails, NULL is returned.

`name`
: input value; points to either the official name or an alias of the host whose Internet address is to be found.

`af`
: input value; an integer that sets the address type searched for by the function and returned by the function. `af` is either AF_INET (IPv4) or AF_INET6 (IPv6).

Errors

`gethostbyname2` returns NULL to indicate an error. In this case, the global variable `h_errno` contains one of these error codes (as defined in netdb.h):

`HOST_NOT_FOUND`
: The specified host was not found. This is the only possible value if the resolver code has been disabled.

`TRY_AGAIN`
: The local server did not receive a response from an authoritative server. Try again later.

NO_RECOVERY
An error has occurred from which there is no recovery.

Example

The example makes a call to gethostbyname2 by passing the host-name and address family as arguments. If an answer is found, a pointer to the hostent structure is returned and stored in hp. NULL is returned if no answer is found.

```
int af;
char *name;
struct hostent *hp;
hp = gethostbyname2(name, af);
```

Usage Guidelines

- The parameter name passed to the gethostbyname2 function is case-sensitive.
- The hostent structure is statically declared. Subsequent calls to gethostbyname2 replace the existing data in the hostent structure.

gethostid

The gethostid function gets the host ID of the local host. The host address returned corresponds to the address returned in the SCF command INFO PROCESS (or its programmatic equivalent).

C Synopsis

```
#include <netdb.h>

id = gethostid ();
     int id;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

id := gethostid ();
     INT(32)  id;
```

id
return value; an integer, assigned by the system administrator, which uniquely identifies the host to the Internet. Often it is the local address part of the Internet address assigned to the host. This is the return value.

Errors

No errors are returned for this function.

gethostname

The gethostname function gets the official name by which the local host is known to the Internet. The hostname returned corresponds to the hostname returned in the SCF command INFO PROCESS (or its programmatic equivalent).

C Synopsis

```
#include <netdb.h>

error = gethostname (buffer, buffer_length);

         int error;
```

```
          char buffer [];
          socklen_t buffer_length;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKPROC
?NOLIST, SOURCE SOCKDEFT

error := gethostname (buffer, buffer_length);

          INT(32)      error;
          STRING .EXT buffer;
          INT(32)      buffer_length;
```

error
: return value; if the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 114).

buffer
: return value; a character array in which the official name of the local host is returned. The name returned is a null-terminated character string (for example, “medlab\0”).

buffer_length
: input value; the size of *buffer*.

Errors

If an error occurs, the external variable *errno* is set to the following value:

EINVAL	An invalid argument was specified.

getipnodebyaddr

The getipnodebyaddr function searches host entries until a match with the *src* is found. (This function is supported for NonStop TCP/IP only.)

The getipnodebyaddr function returns a pointer to a hostent struct whose members specify data from a name server specified in the resconf or hosts files.

NOTE: The C synopsis is given in ANSI C format rather than the pre-ANSI C formats of the other library routines because the only NonStop servers you can use these routines on all support ANSI C. (ANSI C format defines the function and the arguments in the same line rather than using an assign statement and defining the arguments underneath.)

C Synopsis

```
#include <sys/socket.h>

#include <netdb.h>
struct hostent *getipnodebyaddr (const void *src, socklen_t len,
int af, int *error_ptr);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

return_value := getipnodebyaddr(src, len, af, error_ptr);

     INT(32)     return_val;
     STRING .EXT src;
     INT(32)     len;
     INT(32)     af;
```

```
INT .EXT    error_ptr;
```

return_value
is a pointer to a structure of type `hostent`.

src
input value; a pointer to an IP address for which the hostname should be returned; the address specified should be in binary format and network order.

len
input value; the length of the IP address: 4 octets for `AF_INET` or 16 octets for `AF_INET6`.

af
input value; member of address family `AF_INET` or `AF_INET6`.

error_ptr
input and return value; a pointer to the integer containing an error code, if any.

Usage Guidelines

- `getipnodebyaddr` provides the same functionality as gethostbyaddr, host_file_gethostbyaddr but is protocol-independent.
- The `getipnodebyaddr` function has the same arguments as the IPv4 `gethostbyaddr` function but adds an error number value. The `error_num` value is returned to the caller with the appropriate error code to support thread safe error code returns.
- A thread-safe environment must be used with the `getipnodebyaddr` function.
- The `getipnodebyaddr` function processes IPv4-compatible IPv6 addresses as follows:
  1. When *af* is `AF_INET6` and *len* equals 16, and when the IPv6 address is an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address, the function:
     a. Skips the first 12 bytes of the IPv6 address.
     b. Sets *af* to `AF_INET`.
     c. Sets *len* to four.
  2. If *af* is `AF_INET`, the function looks up the name for the given IPv4 address.
  3. If *af* is AF_INET6, the function looks up the name for the given IPv6 address.
- A successful function call returns a pointer to the `hostent` structure that contains the hostname. The structure returned also contains the values used for *src* and *addr_family*., possibly modified as described in the preceeding usage guideline.
- Information returned by `getipnodebyaddr` is dynamically allocated. The information is the `hostent` structure and the data areas pointed to by the members of the `hostent` structure are all dynamically allocated. To return the memory to the system, call the `freehostent` function.

Errors

An unsuccessful function returns NULL pointer and one of the following nonzero values for the *error_ptr*:

HOST_NOT_FOUND	The specified address is not valid.
NO_RECOVERY	A server failure occurred. This is a nonrecoverable error.
TRY_AGAIN	An error occurred that might have been caused by a transient condition. A later retry might succeed.

getipnodebyname

The `getipnodebyname` function gets host information based on the hostname. This function is protocol-independent. (This function is supported for Parallel Library TCP/IP only.)

NOTE: The C synopsis is given in ANSI C format rather than the pre-ANSI C formats of the other library routines because the only NonStop servers you can use these routines on all support ANSI C. (ANSI C format defines the function and the arguments in the same line rather than using an assign statement and defining the arguments underneath.)

C Synopsis

```
#include <netdb.h>
struct hostent *getipnodebyname (const char *name, int af,
int flags, int *error_ptr);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

 return_val := getipnodebyname(name, af, flags, error_ptr);

      INT (320     return_val;
      STRING .EXT  name;
      INT(32)      af;
      INT(32)      flags;
      INT .EXT     error_ptr;
```

return_val
is a pointer to a structure of type `hostent`.

name
input value; a pointer to a node name or numeric address string, such as an IPv4 dotted-decimal address or an IPv6 hexadecimal address.

af
input value; an integer that sets the address type searched for by the function and returned by the function. *af* is either `AF_INET` (IPv4) or `AF_INET6` (IPv6).

flags
input value; an integer that specifies the conditions for returning an address: IPv6-only, IPv4-mapped if no IPv6 address found, or return an address only if the remote node name has at least one IP address configured. See ai_flags underaddrinfo (page 64) for the `ai_flags` values.

error_ptr
input and return value; a pointer to the error code returned by the `getipnodebyname` function. *error_num* is set to one of the following values:

`HOST_NOT_FOUND`	The specified host was not found.
`TRY_AGAIN`	A temporary, and possibly transient, error occurred, such as a failure of a server to respond.
`NO_RECOVERY`	An unexpected server failure occurred which cannot be recovered.
`NO_ADDRESS`	The specified hostname is valid, but the host does not have an IP address. Another type of request to the name server for the domain might return an error.

Example

The address pointed to by *hp*, which is already in network order, can be used directly in a `sockaddr_in` or `sockaddr_in6` structure, as in the following example:

```
struct sockaddr_in sin;
struct hostent *hp;
 ...
if ((hp = getipnodebyname (nameptr, AF_INET, AI_PASSIVE,
&error_num)) != (struct hostent *)
      NULL) {

    memmove ((char *)&sin.sin_addr.s_addr,
             (char *)hp -> h_addr,
             (size_t) hp -> h_length );
  } ...
```

Usage Guidelines

- The `getipnodebyname` function searches host entries sequentially until a match with the *`name`* argument occurs.
- The `geipnodebyname` function returns a pointer to a structure of type `hostent` whose members specify data obtained from a name server specified in the RESCONF file or from fields of a record line in the network hostname database file.
- `getipnodebyname` provides the same functionality as gethostbyname, host_file_gethostbyname but is protocol-independent.
- A thread-safe environment must be used with the `getipnodebyname` function.

Errors

An unsuccessful function returns a pointer (*`error_ptr`*) to one of the following values:

HOST_NOT_FOUND	The specified name is not a valid hostname or alias.
NO_ADDRESS	The server recognized the request and the name specified but no address is available.
NO_RECOVERY	A server failure occurred. This is a nonrecoverable error.
TRY_AGAIN	An error occurred that might have been caused by a transient condition. A later retry might succeed.

getnameinfo

The `getnameinfo` function translates a protocol-independent host address to hostname. This function uses a socket address to search for a hostname and service name. Given a binary IPv4 or IPv6 address and port number, this function returns the corresponding `hostname` and `service name` from a name resolution service. (This function is supported for NonStop TCP/IPv6 only.)

NOTE: The C synopsis is given in ANSI C format rather than the pre-ANSI C formats of the other library routines because the only NonStop servers you can use these routines on all support ANSI C. (ANSI C format defines the function and the arguments in the same line rather than using an assign statement and defining the arguments underneath.)

C Synopsis

```
#include <netdb.h>
int getnameinfo(const struct sockaddr *sa,socklen_t salen,char
*host,socklen_t hostlen,char *serv,socklen_t servlen,int flags);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := getnameinfo(sa, salen, host, hostlen, serv, servlen, flags);

      INT(32)       error;
```

```
INT .EXT      sa(sockaddr);
INT(32)       salen;
STRING .EXT   host;
INT(32)       hostlen;
STRING .EXT   serv;
INT(32)       servlen;
INT(32)       flags;
```

error
return value; if the call is successful, a 0 (zero) is returned. If the call is not successful, —1 is returned. If the call failed, the external variable *error* is set as indicated in Errors (page 119).

sa
input value; points to the `sockaddr_in` or `sockaddr_in6` struct containing the IP address and port number.

salen
input value; specifies the length of the *sa* argument.

host
input and return value; contains the hostname associated with the IP address or the numeric form of the host address (if the flags value `NI_NUMERICHOST` is used).

hostlen
input value; specifies the size of the *host* buffer to receive the returned value. If you specify 0 (zero), no value is returned for *host*. Otherwise, the value returned is truncated as necessary to fit the specified buffer.

serv
input and return value; contains either the service name associated with the port number or the numeric form of the port number (if the flags value of `NI_NUMERICSERV` is used).

servlen
input value; specifies the size of the *serv* buffer to receive the returned value. If you specify 0 (zero), no value is returned for *serv*. Otherwise, the value returned is truncated as necessary to fit the specified buffer.

flags

`NI_NOFQDN`
input value; specifies to return only the hostname part of the fully qualified domain name (FQDN) for local hosts. If you omit this flag, the function returns the host's fully qualified (canonical) domain name.

`NI_NUMERICHOST`
specifies to return the numeric form of the host address instead of the hostname.

`NI_NAMEREQD`
specifies to return an error if the hostname is not found in the DNS.

`NI_NUMERICSERV`
specifies to return the numeric port number instead of the service name.

`NI_DGRAM`
specifies to return only ports configured for a UDP service. This flag is required for ports that use different services for UDP and TCP.

Usage Guidelines

- By default, this function returns the hostname's fully qualified domain name.
- This function, along with getipnodebyaddr, are protocol-independent replacements for gethostbyaddr, host_file_gethostbyaddr. `getnameinfo` provides extra functionality beyond what getipnodebyaddr provides because it handles both the host's address and port number.
- Appropriate use of this function can eliminate calls to getservbyport and at the same time provide protocol independence.

Example

The following programming example calls the `getnameinfo` routine to get a hostname's fully qualified domain name.

```
#include <socket.h>
#include <netdb.h>

{

...

error = getnameinfo((struct sockaddr *)sin,
   addrlen,hname,sizeof(hname), sname,
   sizeof(sname),NI_NUMERICHOST|NI_NUMERICSERV);
if(error)
   fprintf(stderr, "getnameinfo: %s\n", gai_strerror(error));
}
```

Errors

Upon successful completion, this function returns 0 (zero) and the requested values are stored in the buffers specified for the call. Otherwise, the value returned is nonzero and `errno` is set to indicate the error (only when the error is `EAI_SYSTEM`). See the error codes described in *ecode* .

getnetbyaddr

The `getnetbyaddr` function gets the name of the network corresponding to the specified network address.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

net_entry_ptr = getnetbyaddr (net_addr, type);

      struct netent *net_entry_ptr;
      unsigned long net_addr;

      int type;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

net_entry_ptr := getnetbyaddr (net_addr, type);

      INT(32)  net_entry_ptr;
      INT(32)  type;
```

```
      INT(32)    net_addr;
```

net_entry_ptr
return value; points to a structure (based on the `netent` structure) that contains all the required information on the specified network. This is the return value.

If the lookup fails (for instance, if the specified network address is invalid, if no `NETWORKS` file exists, if the `NETWORKS` file could not be opened, or if no matching network entry is found in the `NETWORKS` file), NULL is returned.

net_addr
input value; the network address by which the network is to be found. Use the `inet_netof` function to obtain the network portion of an Internet address.

type
input value; the type of address specified. Its value must be `AF_INET` or `AF_INET6`.

Errors

No errors are returned for this function.

Usage Guideline

This call requires the presence of a `NETWORKS` file providing information on the networks accessible from this host. The format of this file is described in the *TCP/IPv6 Configuration and Management Manual*.

getnetbyname

The `getnetbyname` function gets the network number of the network with the specified network name.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

net_entry_ptr = getnetbyname (net_name);

      struct netent *net_entry_ptr;
      char *net_name;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

net_entry_ptr := getnetbyname (net_name);

      INT(32)  net_entry_ptr;
      STRING .EXT net_name;
```

net_entry_ptr
return value; points to a structure (based on the `netent` structure) that contains all the required information on the specified network. This is the return value.

If the lookup fails (for instance, if the specified name is invalid, if no `NETWORKS` file exists, if the `NETWORKS` file could not be opened, or if no matching network entry is found in the `NETWORKS` file), NULL is returned.

net_name
input value; a null-terminated character string that contains the network name.

Errors

No errors are returned for this function.

Usage Guidelines

- This call requires the presence of a NETWORKS file providing information on the networks accessible from this host. The format of this file is described in the *TCP/IPv6 Configuration and Management Manual*.
- The parameters passed to the getnetbyname function are case-sensitive.
- The netent structure is statically declared. Subsequent calls to getnetbyname replace the existing data in the netent structure.

getpeername, getpeername_nw

The getpeername and getpeername_nw functions get the address and port number of the remote host to which the specified socket is connected.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

error = getpeername (socket, address_ptr, address_len_ptr);

error = getpeername_nw (socket, address_ptr,
                    address_len_ptr, tag);

        int error, socket, *address_len_ptr;
        struct sockaddr *address_ptr;

        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := getpeername (socket, address_ptr, address_len_ptr);

error := getpeername_nw (socket, address_ptr,
                    address_len_ptr, tag);

        INT(32)       socket,
               .EXT address_len_ptr,
               .EXT address_ptr (sockaddr_in);
        INT(32) tag;
```

error
return value; if the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 122).

socket
input value; specifies the socket number for the socket, as returned by the call to socket or socket_nw.

address_ptr
input and return value; points, on return, to the address and port number of the remote socket to which this socket is connected.

address_len_ptr
input and return value; maintained only for compatibility and should point to a value indicating the size in bytes of the structure (the remote address and port number) pointed to by *address_ptr*.

tag
input value; the *tag* parameter to be used for the nowait operation initiated by getpeername_nw.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

ENOTCONN	The specified socket was not connected.
EINVAL	One of the specified arguments was invalid.

Usage Guidelines

- Use getpeername on a socket created for waited operations, or getpeername_nw on a socket created for nowait operations. The operation initiated by getpeername_nw must be completed with a call to the AWAITIOX procedure.
- Complete the operation initiated by getpeername_nw must be with a call to the Guardian AWAITIOX procedure.
- If an unconnected socket is specified in a call to either the getpeername or getpeername_nw, the function fails. This is typical of socket implementations.
- Declare the *address_ptr* variable as struct sockaddr_in6 * for IPv6 use or as struct sockaddr_storage * for protocol-independent use. In C, when you make the call, cast the variable to sockaddr. (See the IPv6 example.)

See Nowait Call Errors (page 86) for information on error checking.

See Data Structures (page 63) for information about struct sockaddr *.

getprotobyname

The getprotobyname function gets the protocol number of the protocol with the specified name.

C Synopsis

```
#include <netdb.h>

proto_entry_ptr = getprotobyname (proto_name_ptr);

      struct protoent *proto_entry_ptr;
      char *proto_name_ptr;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

proto_entry_ptr := getprotobyname (proto_name_ptr);

      INT(32)  proto_entry_ptr;
      STRING .EXT proto_name_ptr;
```

proto_entry_ptr
return value; points to a structure (based on the protoent structure) that contains all the information available about the specified protocol. This is the return value.

If the lookup fails, NULL is returned.

proto_name_ptr
input value; points to a null-terminated character string that contains the protocol name.

Errors

No errors are returned for this function.

Usage Guidelines

- This call requires the presence of a PROTOCOL file providing information on the available protocols. The information in the default PROTOCOL file is given in Appendix A (page 241). The format of this file is described in the *TCP/IPv6 Configuration and Management Manual*.
- The parameters passed to the getprotobyname function are case-sensitive.
- The protoent structure is statically declared. Subsequent calls to getprotobyname replace the existing data in the protoent structure.

Example

The following programming example makes a call to get information on the ICMP protocol (identified as icmp in the PROTOCOL file):

```
#include <netdb.h>
...
struct protoent *proto;
...
if ((proto = getprotobyname("icmp")) == NULL) {
     fprintf(stderr, "icmp: unknown protocol\n");
     exit (1);
}
/* Call succeeded. Information about icmp is in
 * the proto structure.
 */
```

getprotobynumber

The getprotobynumber function gets the protocol name of the protocol with the specified number.

C Synopsis

```
#include <netdb.h>

proto_entry_ptr = getprotobynumber (proto);

      struct protoent *proto_entry_ptr;
      int proto;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

proto_entry_ptr := getprotobynumber (proto);

      INT(32)  proto_entry_ptr;
      INT proto;
```

proto_entry_ptr
return value; points to a structure (based on the protoent structure) that contains all the information available about the specified protocol. This is the return value.

If the lookup fails, NULL is returned.

proto
input value; the Internet protocol number of the protocol.

Errors

No errors are returned for this function.

Usage Guidelines

- This call requires the presence of a PROTOCOL file providing information on the available protocols. The information in the default PROTOCOL file is given in Appendix A (page 241). The format of this file is described in the *TCP/IPv6 Configuration and Management Manual*.
- The protoent structure is statically declared. Subsequent calls to getprotobynumber replace the existing data in the protoent structure.

Example

The following programming example makes a call to get information on the ICMP protocol (identified as icmp in the PROTOCOL file) by specifying its number:

```
#include <netdb.h>
...
struct protoent *proto;
...
if ((proto = getprotobynumber(1)) == NULL) {
     fprintf(stderr, "Proto 1: unknown protocol\n");
     exit (1);
}
/* Call succeeded. Information about icmp is in
 * the proto structure.
 */
```

getservbyname

The getservbyname function gets the port number associated with the specified service.

C Synopsis

```
#include <netdb.h>

serv_entry_ptr = getservbyname (serv_name_ptr, proto_ptr);

      struct servent *serv_entry_ptr;
      char *serv_name_ptr, *proto_ptr;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

serv_entry_ptr := getservbyname (serv_name_ptr, proto_ptr);

      INT(32)       serv_entry_ptr;
      STRING  .EXT serv_name_ptr,
              .EXT proto_ptr;
```

serv_entry_ptr
return value; points to a structure (based on the servent structure) that contains information on the specified service. This is the return value.

If the lookup fails, NULL is returned.

serv_name_ptr
input value; points to a null-terminated character string that contains the service name.

proto_ptr
input value; points to a null-terminated character string that contains the name of the protocol associated with the service.

Errors

No errors are returned for this function.

Usage Guidelines

- This call requires the presence of a `SERVICES` file providing information on the available services. The information in the default `SERVICES` file is given in Table 19 (page 242). The format of this file is described in the *TCP/IPv6 Configuration and Management Manual* and the *Cluster I/O Protocols Configuration and Management Manual*.
- The `servent` structure is statically declared. Subsequent calls to `getservbyname` replace the existing data in the `servent` structure.

getservbyport

The `getservbyport` function gets the name of the service associated with the specified port.

C Synopsis

```
#include <netdb.h>

serv_entry_ptr = getservbyport (port_number, proto_ptr);

      struct servent *serv_entry_ptr;
      char *proto_ptr;
      int port_number;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

serv_entry_ptr := getservbyport (port_number, proto_ptr);

      INT(32)     serv_entry_ptr;
      INT(32)     port_number;
      STRING .EXT proto_ptr;
```

serv_entry_ptr
return value; points to a structure (based on the `servent` structure) that contains information on the specified service. This is the return value.

If the lookup fails, NULL is returned.

port_number
input value; the port number.

proto_ptr
input value; points to a null-terminated character string that contains the name of the protocol associated with the service.

Errors

No errors are returned for this function.

Usage Guidelines

- This call requires the presence of a SERVICES file providing information on the available services. The format of this file is described in the *TCP/IPv6 Configuration and Management Manual* and in the *Cluster I/O Protocols Configuration and Management Manual*.
- The servent structure is statically declared. Subsequent calls to getservbyport replaces the existing data in the servent structure.

getsockname, getsockname_nw

The getsockname and getsockname_nw functions get the address and port number to which a socket is bound.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

error = getsockname (socket, address_ptr, address_len_ptr);

error = getsockname_nw (socket, address_ptr,
                        address_len_ptr, tag);

        int error, socket;
        struct sockaddr *address_ptr;

        int *address_len_ptr;

        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := getsockname (socket, address_ptr, address_len_ptr);

error := getsockname_nw (socket, address_ptr,
                         address_len_ptr, tag);

        INT(32)      error;
        INT(32)      socket,
            .EXT     address_ptr (sockaddr);
        INT .EXT     address_len_ptr;
        INT(32)      tag;
```

error

return value; if the call is successful, a zero is returned. If the call is not successful, —1is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 127).

socket

input value; specifies the socket number for the socket, as returned by the call to socket or socket_nw.

address_ptr

input and return value; on completion, points to the address and port number to which the socket is bound.

If the socket is not bound, the address returned contains a port number of 0 and the Internet address INADDR_ANY.

address_len_ptr
input and return value; maintained only for compatibility and should be a value indicating the size in bytes of the structure (the remote address and port number) pointed to by *address_ptr*.

tag
input value; the *tag* parameter to be used for the nowait operation initiated by getsockname_nw.

Errors

If an error occurs, the external variable *errno* is set to the following value:

EINVAL	An invalid argument was specified.

Usage Guidelines

- Use getsockname on a socket created for waited operations, or use getsockname_nw on a socket created for nowait operations. The operation initiated by getsockname_nw must be completed with a call to the AWAITIOX procedure.
- This function does not return an address when called on an unconnected UDP socket. In addition, this function does not return a port number for an unconnected UDP socket until the first I/O operation on the socket is completed. This is typical of socket implementations.
- Declare the *address_ptr* variable as struct sockaddr_in6 * for IPv6 use or as struct sockaddr_storage * for protocol-independent use. In C, when you make the call, cast the variable to sockaddr. (See the IPv6 example.)

See Chapter 3 (page 62) for information about struct sockaddr *.

See Nowait Call Errors (page 86) for information on error checking.

Examples

INET: the following programming example gets the address and port number to which the socket chan is bound:

```
#include <socket.h>
#include <in.h>
#include <netdb.h>

struct sockaddr_in lcl;
optlen = sizeof(lcl);
if (getsockname(chan,(struct sockaddr *)&lcl, &optlen) < 0)
      perror ("Get socket name failed.");
/* Code to use the address and port number. */
```

INET6: the following programming example gets the address and port number to which the socket chan is bound:

```
#include <socket.h>
#include <in.h>
#include <in6.h>
#include <netdb.h>

struct sockaddr_in6 lcl;
optlen = sizeof(lcl);

/* Notice that the lcl below is cast as sockaddr * as suggested
in the Usage Guidelines */
if (getsockname(chan,(struct sockaddr *)&lcl, &optlen) < 0)
      perror ("Get socket name failed.");
/* Code to use the address and port number. */
```

getsockopt, getsockopt_nw

The `getsockopt` and `getsockopt_nw` functions return the socket options for a socket.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

error = getsockopt (socket, level, optname, optval_ptr,
                    optlen_ptr);

error = getsockopt_nw (socket, level, optname, optval_ptr,
                       optlen_ptr, tag);

        int error, socket, level, optname;
        char *optval_ptr;

        int *optlen_ptr;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := getsockopt (socket, level, optname, optval_ptr,
                     optlen_ptr);

error := getsockopt_nw (socket, level, optname, optval_ptr,
                        optlen_ptr, tag);

        INT(32)      error;
        INT(32)      socket,
                     level,
                     optname;
        STRING .EXT  optval_ptr;
        INT .EXT     optlen_ptr;
        INT(32)      tag;
```

error
: return value; f the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 130).

socket
: input value; specifies the socket number for the socket, as returned by the call to socket or `socket_nw`.

level
: input value; the socket level at which the socket option is being managed. The possible values are:

`SOL_SOCKET`	Socket-level option.
`IPPROTO_TCP`	TCP-level option.
`IPPROTO_IP`	IP-level option.
`IPPROTO_ICMP`	ICMP-level option.
`IPPROTO_UDP`	UDP-level option.
`IPPROTO_RAW`	Raw-socket level option.
user-protocol	Option for a user-defined protocol above IP, such as PUP.

user-protocol can be any protocol number other than the numbers for TCP, UDP, IP, ICMP, and raw. Appendix A (page 241), lists the protocol numbers.

optname

input value; the socket option name.

When *level* is SOL_SOCKET, the possible values are:

SO_BROADCAST	Get the value of the SO_BROADCAST flag. See setsockopt, setsockopt_nw (page 184) for details.
SO_ERROR	Get the error status and clear the socket error. This option applies only to the getsockopt function.
SO_TYPE	Get the socket type. This option applies only to the getsockopt and getsockopt_nw functions.

SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_RAW	Raw socket

SO_DONTROUTE	Get the value of the SO_DONTROUTE flag. See setsockopt, setsockopt_nw (page 184) for details.
SO_REUSEADDR	Get the value of the SO_REUSEADD flag. See setsockopt, setsockopt_nw (page 184) for details.
SO_LINGER	Get the value of the SO_LINGER flag. See setsockopt, setsockopt_nw (page 184) for details.
SO_KEEPALIVE	Get the value of the SO_KEEPALIVE flag. See setsockopt, setsockopt_nw (page 184) for details.
SO_OOBINLINE	Get the value of the SO_OOBINLINE flag. See setsockopt, setsockopt_nw (page 184) for details.
SO_SNDBUF	Get the value of the SO_SNDBUF flag. See setsockopt, setsockopt_nw (page 184) for details.
SO_RCVBUF	Get the value of the SO_RCVBUF flag. See setsockopt, setsockopt_nw (page 184) for details.

When *level* is IPPROTO_IP or IPPROTO_IPV6, the value is:

IP_OPTIONS	Get the value of the IP_OPTIONS flag. See setsockopt, setsockopt_nw (page 184) for details.
IP_MULTICAST_IF or IPV6_MULTICAST_IF	Get the multicast interface IP address. See setsockopt, setsockopt_nw (page 184) for details.
IP_MULTICAST_TTL or IPV6_MULTICAST_HOPS	Get the time-to-live for the multicast datagram. setsockopt, setsockopt_nw (page 184) for details.
IP_MULTICAST_LOOP or IPV6_MULTICAST_LOOP	Get the value of the IP_MULTICAST_LOOP flag. See setsockopt, setsockopt_nw (page 184) for details.
IPV6_V6ONLY	AF_INET6 sockets are restricted to IPv6–only communication.

When *level* is IPPROTO_TCP, you should include the tcp.h file. The value is:

TCP_NODELAY	Get the value of the TCP_NODELAY flag. See setsockopt, setsockopt_nw (page 184) for details.
TCP_SACKENA	Get the value of the TCP_SACKENA flag. See setsockopt, setsockopt_nw (page 184) for details.
TCP_MINRXMT	Get the value of the TCP_MINRXMT flag. See setsockopt, setsockopt_nw (page 184) for details.

TCP_MAXRXMT	Get the value of the TCP_MAXRXMT flag. See setsockopt, setsockopt_nw (page 184) for details.
TCP_RXMTCNT	Get the value of the TCP_RXMTCNT flag. See setsockopt, setsockopt_nw (page 184) for details.
TCP_TOTRXMTVAL	Get the value of the TCP_TOTRXMTVA flag. See setsockopt, setsockopt_nw (page 184) for details.

When *level* is a user-defined protocol above IP, the possible values are defined by the protocol.

optval_ptr
input and return value; points to the value of the socket option specified by *optname*, which is passed to the level specified in *level*. Types and lengths of getsockopt socket option values are described in setsockopt, setsockopt_nw (page 184).

optlen_ptr
input and return value; points, on return from the getsockopt routine, to the length of the value pointed to by *optval_ptr*. The value is zero for the getsockopt_nw routine because this parameter has no meaning for this routine; the length is not known until the AWAITIOX call is completed.

tag
input value; the *tag* parameter to be used for the nowait operation initiated by getsockopt_nw.

Errors

If an error occurs, the external variable *errno* is set to the following value:

ENOPROTOOPT	The specified option is unknown to the protocol.

Usage Guidelines

- Use getsockopt on a socket created for waited operations, or getsockopt_nw on a socket created for nowait operations. The operation initiated by getsockopt_nw must be completed with a call to the AWAITIOX procedure.
- The operation initiated by getsockopt_nw must be completed with a call to the Guardian AWAITIOX procedure.

See Nowait Call Errors (page 86) for information on checking errors.

Examples

See Client and Server Programs Using UDP (page 219) for examples that call the getsockopt function.

if_freenameindex

The if_freenameindex function frees dynamic memory allocated by the if_nameindex function. (This function is supported for NonStop TCP/IP only.)

NOTE: The C synopsis is given in ANSI C format rather than the pre-ANSI C formats of the other library routines because the only NonStop servers you can use these routines on all support ANSI C. (ANSI C format defines the function and the arguments in the same line rather than using an assign statement and defining the arguments underneath.)

C Synopsis

```
#include <if.h>
```

```
#include <netdb.h>

   void if_freenameindex (struct if_nameindex *ptr);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

if_freenameindex(ptr);

       INT .EXT ptr;
```

ptr
input value; specifies the address pointer returned by the `if_nameindex` function for which storage should be returned to the system.

Errors

This function does not return a value. Upon successful completion, all dynamic storage associated with the interface index has been returned to the system.

Usage Guidelines

When an interface (subnet) is created, that interface is assigned a unique number called an interface index. The interface index identifies the interface used to send or receive multicast datagrams. Interface index numbers start with 1.

The `if_freenameindex` function is one of four functions used to manage interface indexes.

Examples

The end of the array of structures is indicated by a structure that has an `if_index` of 0 and an `if_name` of NULL. The memory used for this array of structures along with the interface names pointed to by the `if_name` members is obtained dynamically using the `if_nameindex` function as follows:

```
ifnameindex = if_nameindex();
if ( ifnameindex == NULL) {
  perror("if_nameindex");
}
  freep = ifnameindex;
while (ifnameindex->if_index) {
printf("if_nameindex: index, name: %i, %s\n",
ifnameindex->if_index, ifnameindex >if_name);
ifnameindex++;
}
if_freenameindex(freep);
```

if_indextoname

The `if_indextoname` function maps an interface index to its corresponding name. (This function is supported for Parallel Library TCP/IP only.)

NOTE: The C synopsis is given in ANSI C format rather than the pre-ANSI C formats of the other library routines because the only NonStop servers you can use these routines on all support ANSI C. (ANSI C format defines the function and the arguments in the same line rather than using an assign statement and defining the arguments underneath.)

C Synopsis

```
#include <netdb.h>

char *if_indextoname(unsigned int ifindex, char *ifname);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

name^ptr = if_indextoname(ifindex, char *ifname);

        INT(32)      name^ptr;
        INT(32)      ifindex;
        STRING .EXT ifname;
```

name^ptr
: return value; a pointer to the interface name string. If there is no interface corresponding to the specified index, `NULL` is returned, and error is as described in Errors (page 132).

ifindex
: input value; specifies the index to be mapped to an interface name.

ifname
: input value; specifies the buffer to receive the mapped name. The buffer must be at least `IF_NAMESIZE` bytes long; `IF_NAMESIZE` is defined in the header file `in.h`.

Errors

Upon successful completion, this function returns a pointer to the character string buffer containing the mapped name. Otherwise, this function returns `NULL` and `errno` is set to indicate the following errors.

`EINVAL`	An invalid argument was specified.
`ENOMEM`	Either no memory is available to complete the request or a system error occurred.
`ENXIO`	No interface corresponds to the index specified by the *ifindex* parameter.

Usage Guidelines

When an interface (subnet) is created, that interface is assigned a unique number called an interface index. The interface index identifies the interface used to send or receive multicast datagrams. Interface index numbers start with 1.

The `if_indextoname` function is one of four functions used to manage interface indexes.

Examples

```
cp = if_indextoname(if_index, sn);
if (cp==NULL){
  perror("No interface name matching interface index");
  exit(1);
}
```

if_nameindex

The `if_nameindex` function gets all interface names and indexes. This function returns a pointer to an array of `if_nameindex` structures. See if_nameindex (page 132) for a definition of the `if_nameindex` structure. (This function is supported for NonStop TCP/IPv6 only.)

NOTE: The C synopsis is given in ANSI C format rather than the pre-ANSI C formats of the other library routines because the only NonStop servers you can use these routines on all support ANSI C. (ANSI C format defines the function and the arguments in the same line rather than using an assign statement and defining the arguments underneath.)

C Synopsis

```
#include <if.h>
#include <in.h>
#include <in6.h>
#include <netdb.h>

   struct if_nameindex *if_nameindex(void);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

return_value = if_nameindex();

 INT(32) return_value;
```

Errors

Upon successful completion, this function returns a pointer to an array of `if_nameindex` structures. The end of the array is a structure that has an *`if_index`* value of 0 (zero) and an *`if_name`* value that is NULL pointer.

Otherwise, this function returns `NULL`.

Usage Guidelines

When an interface (subnet) is created, that interface is assigned a unique number called an interface index. The interface index identifies the interface used to send or receive multicast datagrams. Interface index numbers start with 1.

The `if_nameindex` function is one of four functions used to manage interface indexes.

NOTE: Memory is dynamically allocated for the array of structures returned by this function and for the interface names pointed to by the if_name members of the structures. Use the `if_freenameindex` function to return this memory to the system when it is no longer needed.

Examples

```
ifnameindex = if_nameindex();
if (ifnameindex == NULL){
  perror("if_nameindex failed");
}
  freep = ifnameindex;
  while (ifnameindex->if_index){
  printf("if_nameindex: index, name: %i, %s\n",
ifnameindex->if_index, ifnameindex -> if_name);
ifnameindex++;
}

if_freenameindex(freep);
```

if_nametoindex

The `if_nametoindex` function maps an interface name to its corresponding index. (This function is supported for NonStop TCP/IP only.)

C Synopsis

```
#include <netdb.h>

  unsigned int if_nametoindex(const char *ifname);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC
```

```
index = if_nametoindex(ifname);

      INT(32)      index;
      STRING .EXT ifname;
```

index
: return value; upon successful completion, `if_nametoindex` returns the interface index corresponding to the interface name specified in *ifname*. Otherwise, this function returns 0 (zero).

ifname
: input value; points to a buffer that holds the name of the interface (subnet) to be mapped to an index number. The name specified cannot be larger than `IFNAMSIZ`, as defined in the if.h header file.

Usage Guidelines

When an interface (subnet) is created, that interface is assigned a unique number called an interface index. The interface index identifies the interface used to send or receive multicast datagrams. Interface index numbers start with 1.

The `if_nametoindex` function is one of four functions used to manage interface indexes.

Example

```
if_index = if_nametoindex(&subnetname);
if (if_index <= 0){
  perror("Interface name not found");
exit(1);
}
```

inet_addr

The `inet_addr` function converts an address format from dotted-decimal format to binary format. This call is for INET operations. For protocol-independent applications, see inet_pton (page 139).

C Synopsis

```
#include <netdb.h>

l_addr = inet_addr (addr_ptr);

         unsigned long l_addr;
         char *addr_ptr;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

l_addr := inet_addr (addr_ptr);

      INT(32)      l_addr ;
      STRING .EXT   addr_ptr;
```

l_addr
: return value; the Internet address in binary format. This value is the return value. This address can be copied directly into the structure `sockaddr_in`.

addr_ptr
: input value; points to an Internet address in dotted-decimal format.

Errors

`0xffffffff` is returned if the character string that is passed is not an Internet address.

Example

See UDP Client Program (page 219) for an example that calls `inet_addr`.

inet_lnaof

The `inet_lnaof` function breaks apart an INET Internet address and returns the local address portion.

C Synopsis

```
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

l_addr = inet_lnaof (addr);

         unsigned long l_addr;
         struct in_addr addr;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

l_addr := inet_lnaof (addr);

      INT(32)  l_addr ;
      INT.EXT   addr(in_addr);
```

l_addr
return value; the local address portion of the Internet address. This is the return value.

addr
input value; a 4-byte Internet address.

Errors

No errors are returned for this function.

inet_makeaddr

The `inet_makeaddr` function combines an INET family network address and a local address to create an INET family Internet address.

C Synopsis

```
#include <in.h>
#include <in6.h>
#include <netdb.h>

inaddr = inet_makeaddr (net, lna);

         unsigned long net, lna;
         struct in_addr inaddr;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

inaddr := inet_makeaddr (net, lna);
```

```
INT(32) inaddr, net, lna;
```

inaddr
return value; the corresponding 4-byte Internet address. This is the return value.

net
input value; the network address portion of the Internet address.

lna
input value; the local address portion of the Internet address.

Errors

No errors are returned for this function.

inet_netof

The `inet_netof` function breaks apart an INET family Internet address and returns the network address portion.

C Synopsis

```
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

net = inet_netof (addr);

      unsigned long net;
      struct in_addr addr;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

net := inet_netof (addr);

      INT(32) net;
      INT .EXT addr(in_addr);
```

net
return value; the network address portion of the Internet address. This is the return value.

addr
input value; a 4-byte Internet address.

Errors

No errors are returned for this function.

inet_network

The `inet_network` function converts an INET family address from dotted-decimal format to binary format and returns the network address portion.

C Synopsis`#include <in.h>`

```
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

l_addr = inet_network (addr_ptr);
```

```
        unsigned long l_addr;
        char *addr_ptr;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

l_addr := inet_network (addr_ptr);

      INT(32) l_addr ;
      STRING .EXT addr_ptr;
```

l_addr
return value; the network address portion of the Internet address. This is the return value.

addr_ptr
input value; points to an Internet address in dotted-decimal format.

Errors

No errors are returned for this function.

inet_ntoa

The `inet_ntoa` function converts an address from binary format to dotted-decimal format. This library routine is for INET applications. For protocol-independent applications, see inet_ntop (page 138).

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h>
#include <netdb.h>

asc_ptr = inet_ntoa (in);

      struct in_addr in;
      char *asc_ptr;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

asc_ptr := inet_ntoa (in);

      INT(32)   asc_ptr;
      INT .EXT in(in_addr);
```

asc_ptr
return value; points to a null-terminated character string containing the Internet address in dotted-decimal format. All numbers are expressed in decimal base. This is the return value.

in
input value; a 4-byte Internet address.

Errors

No errors are returned for this function.

inet_ntop

The `inet_ntop` function converts an IPv6 or IPv4 binary address to a character string. (This function is supported for Parallel Library TCP/IP only.)

NOTE: The C synopsis is given in the ANSI C format rather than the pre-ANSI C formats of the other library routines because the only NonStop servers you can use these routines on all support ANSI C. (The ANSI C format defines the function and the arguments in the same line rather than using an assign statement and defining the arguments underneath.)

C Synopsis

```
#include <netdb.h>
const char *inet_ntop(int af,const void *src,char *dst, socklen_t size);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

return_value = inet_ntop(af, src, dst, size);

      INT(32)     return_value;
      INT(32)     af;
      STRING .EXT src;
      STRING .EXT dst;
      INT(32)     size;
```

return_value
is a pointer to the buffer containing the text string if the conversion succeeds, and `NULL` otherwise.

af
input value; specifies the address family for the address to be converted. Valid values are:

`AF_INET`
indicates an IPv4 address

`AF_INET6`
indicates an IPv6 address

src
input value; points to a buffer containing the network byte-ordered INET or INET6 binary address to be converted.

dst
input and return value; specifies the non-NULL address of the location to receive the converted character string.

size
input value; specifies the length of the buffer pointed to by *dst*. Valid values for INET are greater than or equal to 16 bytes and for INET6 are greater than or equal to 46 bytes.

NOTE: The maximum length of an INET address as a text string is defined as `INET_ADDRSTRLEN` in the `in.h` header file. The maximum length of an INET6 address as a text string is defined as `INET6_ADDRSTRLEN` in the in6.h header file.

Errors

Upon successful completion, this function returns a pointer to the dst buffer. Otherwise, this function returns `NULL` and `errno` is set to indicate the error. If any of these conditions occurs, the function sets `errno` to the corresponding value:

EAFNOSUPPORT	The value specified for the *af* parameter is not valid.
ENOSPC	The value specified for the *size* parameter is not valid for the address family.

Usage Guidelines

The `inet_ntop` function is one of two functions that allow you to manage network addresses regardless of the address family.

inet_pton

The `inet_pton` function converts a character string to an IPv6 or IPv4 binary address. (This function is supported for NonStop TCP/IPv6 only.)

NOTE: The C synopsis is given in ANSI C format rather than the pre-ANSI C formats of the other library routines because the only NonStop servers you can use these routines on all support ANSI C. (ANSI C format defines the function and the arguments in the same line rather than using an assign statement and defining the arguments underneath.)

C Synopsis

```
#include <netdb.h>
int inet_pton(int af, const char *src, void *dst);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error = inet_pton(af, src, dst);

        INT  error;
        INT  af;
        STRING .EXT src;
        STRING .EXT dst;
```

af
input value; specifies the address family for the address to be converted. Valid values are:

`AF_INET`
indicates an IPv4 address

`AF_INET6`
indicates an IPv6 address

src
input value; points to the text string version of the address to be converted. This parameter cannot be a null pointer. *src* has one of the following forms:

- IPv4 dotted decimal format as ddd.ddd.ddd.ddd, for example:

  `172.17.201.43`

- IPv6 hexadecimal string format as x:x:x:x:x:x:x:x, for example:

  `1080:0:0:0:8:800:200C:417A`

- Compressed hexadecimal string format that omits zero values, for example:

```
1080::8:800:200C:417A
```

- In mixed form as x:x:x:x:x:x:d.d.d.d, for example:

```
::FFFF:13.1.68.3 as a mapped value, or ::13.1;68.3 as a compatible value.
```

dst

input and return value; receives the converted address in network byte order.

NOTE: The maximum length of an IPv4 address as a text string is defined as INET_ADDRSTRLEN in the in.h header file. The maximum length of an IPv6 address as a text string is defined as INET6_ADDRSTRLEN in the in6.h header file.

Errors

Upon successful completion, this function returns a 1. Otherwise, this function returns:

0	The *dst* parameter specifies an invalid address string.
-1	The *af* parameter specifies an invalid address string.

When -1 is returned, errno is also set.

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

EAFNOSUPPORT	The value specified for the *af* parameter is not valid.

Usage Guidelines

The inet_pton function is one of two functions that allow you to manage network addresses regardless of address family.

lwres_freeaddrinfo

The lwres_freeaddrinfo function frees the memory of one or more addrinfo structures previously created by the lwres_getaddrinfo function. Any dynamic storage pointed to by the structure is also freed. (This function is supported for G06.27 and later G-series RVUs and H06.05 and later H-series RVUs of NonStop TCP/IPv6.)

C Synopsis

```
#include <netdb.h>

void lwres_freeaddrinfo (struct addrinfo *ai);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

lwres_freeaddrinfo (ai);
    INT .EXT ai (addrinfo);
```

ai

input value; specifies the addrinfo structure to be freed.

Usage Guidelines

Call this function once for each structure created by calls to lwres_getaddrinfo before closing a socket. Upon successful completion, lwres_freeaddrinfo does not return a value. The address information structure and associated storage have been returned to the system.

lwres_freehostent

The `lwres_freehostent` function frees the memory of one or more `hostent` structures returned by the `lwres_getipnodebyaddr` or `lwres_getipnodebyname` functions. (This function is supported for G06.27 and later G-series RVUs and H06.05 and later H-series RVUs of NonStop TCP/IPv6.)

C Synopsis

```
#include <netdb.h>

void lwres_freehostent(struct hostent *ptr);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

lwres_freehostent( ptr);
      INT .EXT ptr(hostent);
```

`ptr`
input value; a pointer to the structure `hostent` that has to be freed.

Usage Guidelines

Call this function once for each `hostent` structure returned by the `lwres_getipnodebyaddr` or `lwres_getipnodebyname` functions.

lwres_gai_strerror

The `lwres_gai_strerror` function aids applications in printing error messages based on the `EAI_` codes returned by the `lwres_getaddrinfo` function. The `lwres_gai_strerror` function call returns a pointer to a character string describing the error code passed into the function. (This function is supported for G06.27 and later G-series RVUs and H06.05 and later H-series RVUs of NonStop TCP/IPv6.)

C Synopsis

```
#include <netdb.h>

char * lwres_gai_strerror(int ecode);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

return_value := lwres_gai_strerror ( ecode);
      INT(32) return_value;
      INT ecode;
```

`return_value`
is a pointer to a string described in `ecode`.

`ecode`
input value; specifies one of the following error codes returned by the `lwres_getaddrinfo` function. The returned strings are as follows:

`EAI_ADDRFAMILY`
address family for `hostname` not supported.

EAI_AGAIN
: temporary failure in name resolution.

EAI_BADFLAGS
: invalid value for ai_flags.

EAI_FAIL
: non-recoverable failure in name resolution.

EAI_FAMILY
: ai_family not supported.

EAI_MEMORY
: memory allocation failure.

EAI_NODATA
: no address associated with hostname.

EAI_NONAME
: hostname or servname not provided, or not known.

EAI_SERVICE
: servname not supported for ai_socktype.

EAI_SOCKTYPE
: ai_socktype not supported.

EAI_SYSTEM
: system error returned in errno.

Errors

The message invalid error code is returned if ecode is out of range. ai_flags, ai_family, and ai_socktype are elements of the struct addrinfo used by lwres_getaddrinfo.

Example

The following programming example calls the gai_strerror routine to print error messages:

```
ret = lwres_getaddrinfo(hostname, servname, &hints, &result);

if(ret != 0) {
fprintf(stderr,"%s", lwres_gai_strerror(error));
return -1;
}
```

Usage Guidelines

Call this function to aid in printing human-readable error messages based on the EAI_ error codes returned by the lwres_getaddrinfo function.

lwres_getaddrinfo

The lwres_getaddrinfo function converts hostnames and service names into socket address structures. This function is defined for protocol-independent hostname-to-address translation. It performs the functionality of lwres_gethostbyname but in a more sophisticated manner. (This function is supported for G06.27 and later G-series RVUs and H06.05 and later H-series RVUs of NonStop TCP/IPv6.)

C Synopsis

```
#include netdb.h>
```

```
int lwres_getaddrinfo (const char *hostname, const char
*servname, const struct addrinfo *hints, struct addrinfo **result);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := lwres_getaddrinfo (hostname, servname, hints, result);

    INT error;
    STRING .EXT hostname;
    STRING .EXT servname;
    INT .EXT hints(addrinfo);
    INT .EXT result(addrinfo);
```

`hostname`
input value; specifies a pointer to a character representing one of the following:

- An Internet node hostname.
- An IPv4 address in dotted-decimal format.
- An IPv6 address in hexadecimal format.
- NULL if no hostname requires converting; when NULL is used, either `service` or `hints` must be non-NULL.

`servname`
input value; specifies a pointer to a character representing one of the following:

- A network service name.
- A decimal port number.
- NULL if no service name requires converting; when NULL is used, either hostname or hints must be non-NULL.

`hints`
input value; specifies one of the following:

- A pointer to an `addrinfo` struct for a socket; the format of the `addrinfo` structure is defined in the header file netdb.h.
- `NULL` if no struct is available; when NULL is used, either hostname or service must be non-NULL.

`result`
input and return value; points to a list of `addrinfo` structs upon successful completion (See Usage Guidelines (page 144).)

Errors

`lwres_getaddrinfo` returns zero (0) on success or one of the error codes listed in `lwres_gai_strerror` if an error occurs. If both hostname and service are NULL `lwres_getaddrinfo` returns `EAI_NONAME`.

Example

```
struct addrinfo *res, *ainfo;
struct addrinfo hints;
int ret;
char *hostname, *servname;

/* clear out hints */
memset ((char *)&hints, 0, sizeof(hints));
```

```
hints.ai_socktype = SOCK_STREAM;

ret = getaddrinfo(hostname, servname, &hints, &res);

if (ret != 0) {
fprintf(stderr, "%s not found in name service database\n",
hostname);
exit(1);
}
for (ainfo = res; ainfo != NULL; ainfo = ainfo->ai_next) {
/* Create the socket. */
s = socket(ainfo->ai_family, ainfo->ai_socktype,
ainfo->ai_protocol);

if (connect(s, ainfo->ai_addr, ainfo->ai_addrlen) == -1) {
perror(argv[0]);
fprintf(stderr, "unable to connect\n");
FILE_CLOSE(S);
continue;
}
else
break;
}
```

Usage Guidelines

- This function is a protocol-independent replacement for `lwres_gethostbyname` and `lwres_getipnodebyname`. `lwres_getaddrinfo` provides extra functionality because `lwres_getaddrinfo` handles both the hostname and the service.
- The `lwres_getaddrinfo` function converts hostnames and service names into socket address structures. You allocate a hints structure, initialize it to zero (0), fill in the needed fields, and call this function.
- This function returns, through the result pointer, a linked list of `addrinfo` structures (defined in netdb.h) that you can use with other socket functions.
- The `lwres_freeaddrinfo` function returns the storage allocated by the `lwres_getaddrinfo` function.

lwres_gethostbyaddr

The `lwres_gethostbyaddr` function gets the name of the host that has the specified Internet address and address family. (This function is supported for G06.27 and later G-series RVUs and H06.05 and later H-series RVUs of NonStop TCP/IPv6.)

C Synopsis

```
#include <netdb.h>

host_entry_ptr = lwres_gethostbyaddr(addr, len, type);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

host_entry_ptr := lwres_gethostbyaddr(addr, len, type);
     INT(32) host_entry_ptr;
     STRING .EXT addr;
     INT len, type;
```

host_entry_ptr
: return value; points to a structure (based on the hostent structure) in which information about the specified host is returned. The information includes the official name, aliases, and addresses for the host. If the lookup fails, NULL is returned, and the external variable lwres_h_errno is set as indicated below under Errors.

addr
: input value; points to the Internet address of the host whose name is to be found. The address pointed to is in binary format and network order. (This address is in the same format and order as the return value of the inet_addr function.)

len
: input value; the length of the Internet address pointed to by host_addr_ptr.

type
: input value; the type of address specified: either AF_INET (IPv4) or AF_INET6 (IPv6).

Errors

lwres_gethostbyaddr returns NULL to indicate an error. In this case, the global variable lwres_h_errno contains one of the following error codes as defined in netdb.h:

HOST_NOT_FOUND	The host or address was not found.
TRY_AGAIN	A recoverable error occurred, for example, a timeout. Retrying the lookup may succeed.
NO_RECOVERY	A non-recoverable error occurred.
NO_DATA	The name exists, but has no address information associated with it (or for a reverse lookup, the address information exists but has no name associated with it). The code NO_ADDRESS is accepted as a synonym for NO_DATA for backwards compatibility.

lwres_hstrerror (page 152) translates these error codes into readable error messages.

Example

The example makes a call to lwres_gethostbyaddr by passing the Internet address as an argument. If an answer is found, a pointer to the hostent structure is returned and stored in hp. NULL is returned if no answer is found.

```
char *addr;
int len, type;
struct hostent *hp;
hp = lwres_gethostbyaddr(addr, len, type);
```

Usage Guidelines

The address that is returned in host_entry_ptr can be used directly in a sockaddr_in structure. The address is in network order.

lwres_gethostbyname

The lwres_gethostbyname function gets the Internet address (IPv4) of the host whose name is specified. (This function is supported for G06.27 and later G-series RVUs and H06.05 and later H-series RVUs of NonStop TCP/IPv6.)

C Synopsis

```
#include <netdb.h>

host_entry_ptr = lwres_gethostbyname(name);
```

TAL Synopsis

```
?NOLIST,SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

host_entry_ptr := lwres_gethostbyname(name);
    INT(32) host_entry_ptr;
    STRING .EXT name;
```

`host_entry_ptr`
return value; points to a structure (based on the `hostent` structure) in which information about the specified host is returned. The information includes the official name, aliases, and addresses for the host. If the lookup fails, `NULL` is returned, and the external variable `lwres_h_errno` is set as indicated below under Errors.

`name`
input value; points to either the official name or an alias of the host whose Internet address is to be found.

Errors

`lwres_gethostbyname` returns `NULL` to indicate an error. In this case, the global variable `lwres_h_errno` contains one of the following error codes as defined in netdb.h:

`HOST_NOT_FOUND`	The host or address was not found.
`TRY_AGAIN`	A recoverable error occurred, for example, a timeout. Retrying the lookup may succeed.
`NO_RECOVERY`	A non-recoverable error occurred.
`NO_DATA`	The name exists, but has no address information associated with it (or for a reverse lookup, the address information exists but has no name associated with it). The code `NO_ADDRESS` is accepted as a synonym for `NO_DATA` for backwards compatibility.

lwres_hstrerror (page 152) translates these error codes into readable error messages.

Example

```
char *name;
struct hostent *hp;
hp = lwres_gethostbyname(name);
```

The above example makes a call to `lwres_gethostbyname` by passing the hostname as an argument. If an answer is found, a pointer to the `hostent` structure is returned and stored in `hp`. `NULL` is returned if no answer is found.

Usage Guidelines

- The parameter name passed to the `lwres_gethostbyname` function is case-sensitive.
- The `hostent` structure is statically declared. Subsequent calls to `lwres_gethostbyname` replace the existing data in the `hostent` structure.

lwres_gethostbyname2

The `lwres_gethostbyname2` function gets the Internet address (IPv4 or IPv6) of the host whose name is specified. (This function is supported for G06.27 and later G-series RVUs and H06.05 and later H-series RVUs of NonStop TCP/IPv6.)

C Synopsis

```
#include <netdb.h>

host_entry_ptr = lwres_gethostbyname2(name, af);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

host_entry_ptr := lwres_gethostbyname2(name, af);
     INT(32) host_entry_ptr;
     STRING .EXT name;
     INT af;
```

`host_entry_ptr`
: return value; points to a structure (based on the `hostent` structure) in which information about the specified host is returned. The information includes the official name, aliases, and addresses for the host. If the lookup fails, `NULL` is returned, and the external variable *`lwres_h_errno`* is set as indicated below under Errors.

`name`
: input value; points to either the official name or an alias of the host whose Internet address is to be found.

`af`
: input value; an integer that sets the address type searched for by the function and returned by the function. `af` is either AF_INET (IPv4) or AF_INET6 (IPv6).

Errors

`lwres_gethostbyname2` returns `NULL` to indicate an error. In this case, the global variable `lwres_h_errno` contains one of the following error codes as defined in netdb.h:

`HOST_NOT_FOUND`	The host or address was not found.
`TRY_AGAIN`	A recoverable error occurred, for example, a timeout. Retrying the lookup may succeed.
`NO_RECOVERY`	A non-recoverable error occurred.
`NO_DATA`	The name exists, but has no address information associated with it (or for a reverse lookup, the address information exists but has no name associated with it). The code `NO_ADDRESS` is accepted as a synonym for `NO_DATA` for backwards compatibility.

lwres_hstrerror (page 152) translates these error codes into readable error messages.

Example

The example makes a call to `lwres_gethostbyaddr2` by passing the hostname and address family as arguments. If an answer is found, a pointer to the `hostent` structure is returned and stored in `hp`. `NULL` is returned if no answer is found.

```
int af;
char *name;
struct hostent *hp;
hp = lwres_gethostbyname2(name, af);
```

Usage Guidelines

- The parameter name passed to the `lwres_gethostbyname2` function is case-sensitive.
- The `hostent` structure is statically declared. Subsequent calls to `lwres_gethostbyname2` replace the existing data in the `hostent` structure.

lwres_getipnodebyaddr

The `lwres_getipnodebyaddr` function searches host entries until a match with `src` is found. The `lwres_getipnodebyaddr` function returns a pointer to a `hostent` struct whose members

specify data from a Name Server. (This function is supported for G06.27 and later G-series RVUs and H06.05 and later H-series RVUs of NonStop TCP/IPv6.)

C Synopsis

```
#include <sys/socket.h>
#include <netdb.h>
return_val = lwres_getipnodebyaddr(const
void *src, socklen_t len, int af, int *error_ptr);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

return_val := lwres_getipnodebyaddr( src, len, af, error_ptr);

     INT(32) return_val;
     STRING .EXT src;
     INT(32) len;
     INT af;
     INT .EXT error_ptr;
```

`return_val`
is a pointer to a structure of type `hostent`.

`src`
input value; a pointer to an IP address for which the hostname should be returned; the address specified should be in binary format and network order.

`len`
input value; the length of the IP address: 4 octets for AF_INET or 16 octets for AF_INET6.

`af`
input value; specifies the member of the address family: AF_INET or AF_INET6.

`error_ptr`
input and return value; a pointer to the integer containing an error code, if any.

Errors

If an error occurs, `lwres_getipnodebyaddr` sets `*error_ptr` to an appropriate error code, and the function returns a NULL pointer. The error codes and their meanings are defined in netdb.h:

`HOST_NOT_FOUND`
The specified host was not found.

`TRY_AGAIN`
A temporary, and possibly transient, error occurred, such as a server not responding.

`NO_RECOVERY`
An unexpected server failure occurred which cannot be recovered.

`NO_ADDRESS`
The specified hostname is valid, but the host does not have an IP address. Another type of request to the Name Server for the domain might return an error.

lwres_hstrerror (page 152) translates these error codes to suitable error messages.

Usage Guidelines

`lwres_getipnodebyaddr` provides the same functionality as `lwres_gethostbyaddr`, but is protocol-independent.

A successful function call returns a pointer to the `hostent` structure that contains the hostname. The structure returned also contains the values used for `src` and `address-family`.

lwres_getipnodebyname

The `lwres_getipnodebyname` function gets host information based on the hostname. This function is protocol-independent. (This function is supported for G06.27 and later G-series RVUs and H06.05 and later H-series RVUs of NonStop TCP/IPv6.)

C Synopsis

```
#include <netdb.h>

return_val = lwres_getipnodebyname (const
char * name, int af, int flags, int * error_ptr);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

return_val := lwres_getipnodebyname( name, af, flags, error_ptr);
  INT (320 return_val;
  STRING .EXT name;
  INT af;
  INT flags;
  INT .EXT error_ptr;
```

`return_val`
: is a pointer to a structure of type `hostent`.

`name`
: input value; a pointer to a node name or numeric address string, such as an IPv4 dotted-decimal address or an IPv6 hexadecimal address.

`af`
: input value; an integer that sets the address type searched for by the function and returned by the function. `af` is either AF_INET (IPv4) or AF_INET6 (IPv6).

`flags`
: input value; contains flag bits to specify the types of addresses that are searched for and the types of addresses that are returned. The flag bits are:

  `AI_V4MAPPED`
  : Used with an `af` of AF_INET6, causes IPv4 addresses to be returned as IPv4-mapped IPv6 addresses.

  `AI_ALL`
  : Used with an `af` of AF_INET6, causes all known addresses (IPv6 and IPv4) to be returned. If `AI_V4MAPPED` is also set, the IPv4 addresses are returned as mapped IPv6 addresses.

  `AI_ADDRCONFIG`
  : Causes a return of an IPv6 or IPv4 address only if an active network interface of that type exists. This flag bit is not currently implemented in the BIND 9 Lightweight resolver, and the flag is ignored.

  `AI_DEFAULT`
  : Sets the `AI_V4MAPPED` and `AI_ADDRCONFIG` flag bits.

`error_ptr`
: input and return value; a pointer to the error code returned by the `lwres_getipnodebyname` function.

lwres_hstrerror (page 152) translates these error codes to readable error messages.

Errors

If an error occurs, `lwres_getipnodebyname` and `lwres_getipnodebyaddr` set `*error_ptr` to an appropriate error code, and the function returns a NULL pointer. The error codes and their meanings are defined in netdb.h:

`HOST_NOT_FOUND`	The host or address was not found.
`TRY_AGAIN`	A recoverable error occurred, for example, a timeout. Retrying the lookup may succeed.
`NO_RECOVERY`	A non-recoverable error occurred.
`NO_DATA`	The name exists, but has no address information associated with it (or for a reverse lookup, the address information exists but has no name associated with it). The code `NO_ADDRESS` is accepted as a synonym for `NO_DATA` for backwards compatibility.

lwres_hstrerror (page 152) translates these error codes into readable error messages.

Example

The address pointed to by `hp`, which is already in network order, can be used directly in a `sockaddr_in` or `sockaddr_in6` structure, as in the following example:

```
struct sockaddr_in sin;
struct hostent *hp;

if ((hp = lwres_getipnodebyname (nameptr, AF_INET, AI_PASSIVE,
&error_num)) != (struct hostent *) NULL) {
memmove ((char *)&sin.sin_addr.s_addr, (char *)hp -> h_addr,
(size_t) hp -> h_length );
```

Usage Guidelines

- The `lwres_getipnodebyname` function searches host entries sequentially until a match with the name argument occurs.
- The `lwres_getipnodebyname` function returns a pointer to a structure of type `hostent` whose members specify data obtained from a Name Server.
- The `hostent` structure is statically declared. Subsequent calls to `lwres_gethostbyname` replace the existing data in the `hostent` structure.
- `lwres_getipnodebyname` provides the same functionality as `lwres_gethostbyname`, but is protocol-independent.

lwres_getnameinfo

The `lwres_getnameinfo` function translates a protocol-independent host address to a hostname. This function uses a socket address to search for a hostname and service name. Given a binary IPv4 or IPv6 address and a port number, this function returns the corresponding hostname and service name from a name0resolution service. (This function is supported for G06.27 and later G-series RVUs and H06.05 and later H-series RVUs of NonStop TCP/IPv6.)

`C Synopsis`

```
#include <netdb.h>
int lwres_getnameinfo(const struct sockaddr *sa, socklen_t
salen, char
* host, socklen_t hostlen,char *serv, socklen_t servlen, int flags);
```

`TAL Synopsis`

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := lwres_getnameinfo( sa, salen, host, hostlen, serv, servlen,
flags);
```

```
INT error;
INT .EXT sa(sockaddr);
INT(32) salen;
STRING .EXT host;
INT(32) hostlen;
STRING .EXT serv;
INT(32) servlen;
INT flags;
```

`error`
return value; if the call is successful, a 0 (zero) is returned. If the call is not successful, —1 is returned.

`sa`
input value; points to the `sockaddr_in` or `sockaddr_in6` struct containing the IP address and port number.

`salen`
input value; specifies the length of the `sa` argument.

`host`
input and return value; contains the returned hostname associated with the IP address or the numeric form of the host address (if the flags value `NI_NUMERICHOST` is used).

`hostlen`
input value; specifies the size of the host buffer to receive the returned value. If you specify 0 (zero), no value is returned for host. Otherwise, the value returned is truncated as necessary to fit the specified buffer.

`serv`
input value; contains either the service name associated with the port number or the numeric form of the port number (if the flags value of `NI_NUMERICSERV` is used).

`servlen`
input value; specifies the size of the serv buffer to receive the returned value. If you specify 0 (zero), no value is returned for `serv`. Otherwise, the value returned is truncated as necessary to fit the specified buffer.

`flags`
input value; one of the following:

`NI_NOFQDN`
specifies to return only the hostname part of the fully qualified domain name (FQDN) for local hosts. If you omit this flag, the function returns the host's fully qualified (canonical) domain name.

`NI_NUMERICHOST`
specifies to return the numeric form of the host address instead of the hostname.

`NI_NAMEREQD`
specifies to return an error if the hostname is not found in the DNS.

`NI_NUMERICSERV`
specifies to return the numeric port number instead of the service name.

`NI_DGRAM`
specifies to return only ports configured for a UDP service. This flag is required for ports that use different services for UDP and TCP.

Errors

Upon successful completion, this function returns 0 (zero) and the requested values are stored in the buffers specified for the call. Otherwise, the value returned is nonzero and `errno` is set to indicate the error (only when the error is `EAI_SYSTEM`). See the return values described for lwres_gai_strerror (page 141).

Example

The example calls the `lwres_getnameinfo` routine to get a hostname's fully qualified domain name.

```
error = lwres_getnameinfo((struct sockaddr *)sin,
addrlen, hname, sizeof(hname), sname,
sizeof(sname), NI_NUMERICHOST|NI_NUMERICSERV);
if(error)
ifprintf(stderr, "Error: %s\n", lwres_gai_strerror(error));
```

Usage Guidelines

By default, this function returns the hostname's fully qualified domain name.

This function, along with `lwres_getipnodebyaddr`, is a protocol-independent replacement for `lwres_gethostbyaddr`. `lwres_getnameinfo` provides extra functionality because it handles both the host's address and port number.

lwres_hstrerror

The `lwres_hstrerror` function returns an appropriate string for the error code given by `err_num`. (This function is supported for G06.27 and later G-series RVUs and H06.05 and later H-series RVUs of NonStop TCP/IPv6.)

C Synopsis

```
#include <netdb.h>
const char * lwres_hstrerror(int err_num);
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

ret_val := lwres_hstrerror(err_num)
     INT(32) ret_val;
     INT err_num;
```

`ret_val`
: return value; a pointer to a string described in `err_num`.

`err_num`
: input value; specifies the integer error code.

Errors

The values of the error codes and messages are:

`NETDB_SUCCESS`
: Resolver error 0 (no error).

`HOST_NOT_FOUND`
: Unknown host.

`TRY_AGAIN`
: hostname lookup failure.

NO_RECOVERY
Unknown server error.

NO_DATA
No address associated with hostname.

listen

The `listen` function is provided for compatibility only. In other socket implementations, `listen` sets the maximum connections that are in the queue awaiting acceptance on a socket. In the NonStop TCP/IP, Parallel Library TCP/IP, and NonStop TCP/IPv6 implementations, the maximum pending connections is always 5. A call to `listen` must precede a call to `accept` or `accept_nw`.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

error = listen (socket, queue_length);

        int error, socket, queue_length;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := listen (socket, queue_length);

        INT error,
            socket,
            queue_length;
```

error
return value; always zero because no error can occur.

socket
input value; specifies the socket number for the socket being used to listen for connections (as returned by a call to `socket` or `socket_nw`).

queue_length
input value; specifies the maximum queue length (number of pending connections). This argument is ignored.

Errors

No errors are returned for this function.

Example

See C TCP Server Program (page 217) for examples that call the `listen` function.

recv, recv_nw

The `recv` and `recv_nw` functions receive data on a connected socket.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

nrcvd = recv (socket, buffer_ptr, length, flags);
```

```
error = recv_nw (socket, buffer_ptr, length, flags, tag);

        int nrcvd, socket;
        char *buffer_ptr;
        int length, flags, error;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

nrcvd := recv (socket, buffer_ptr, length, flags);

error := recv_nw (socket, buffer_ptr, length, flags, tag);

        INT(32)      nrcvd,
                     socket;
        STRING .EXT  buffer_ptr;
        INT(32)      length,
                     flags,
                     error;
        INT(32)      tag;
```

nrcvd
return value; the number of bytes received by the `recv` function. This is the return value for recv. A zero length message indicates end of file (`EOF`).

If the call is not successful, —1 is returned and the external variable *errno* is set as indicated below in "Errors."

error
return value; if the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call failed, the external variable *errno* is set as indicated below in "Errors."

socket
input value; specifies the socket number for the socket, as returned by the call to `socket` or `socket_nw`.

buffer_ptr
input and return value; on completion, points to the received data.

length
input value; the size of the buffer pointed to by *buffer_ptr*.

flags
input value; specifies the kind of data to be read and is one or more of the following:

`MSG_OOB`	Read out-of-band data. This corresponds to the TCP URG flag. You should not select this flag for UDP sockets, or the call fail. This constraint is imposed by UDP, which does not support out-of-band data.
`MSG_PEEK`	Read the incoming message without removing it from the input queue.
`0`	No flag; read data normally.

tag
input value; the *tag* parameter to be used for the nowait operation initiated by `recv_nw`. For more information, see Asynchrony and Nowaited Operations (page 34).

Errors

If an error occurs, the return value is set to -1 and the external variable *errno* is set to one of the following values:

EHAVEOOB	There is out-of-band data pending. This must be cleared with a call to recv with the MSG_OOB flag set. (This error does not apply to UDP sockets.)
ENOTCONN	The specified socket was not connected.
ESHUTDOWN	The specified socket was shut down.
ETIMEDOUT	The connection timed out.
ECONNRESET	The connection was reset by the remote host.

Usage Guidelines

Use the following guidelines for the recv and recv_nw functions:

- Use recv on a socket created for waited operations. Use recv_nw on a socket created with the socket_nw call for nowait operations. The operation initiated by recv_nw must be completed with a call to the AWAITIOX procedure.
- To determine the number of characters read from recv_nw, check the third parameter (the count transferred) returned by the AWAITIOX procedure. Refer to the *Guardian Procedure Calls Reference Manual* for details about the AWAITIOX procedure and its parameters.
- recv and recvfrom could wait indefinitely if the network terminates the connection ungracefully, without returning an error code. This is standard TCP/IP behavior. Avoid the wait by calling recv_nw or recvfrom_nw nowait operations, followed by calling AWAITIOX with a timer value of 10 seconds. If the timer expires, call send or sendto from the local host. If the send or sendto call fails, the connection is down.
- The sending side of a connection indicates end-of-file by closing or shutting down its socket. The receiving side recognizes end-of-file when the recv or recvfrom calls have 0 bytes in their *length-of-buffer* field. This is standard practice, not specific to the Guardian socket library implementation. You are responsible for handling this condition.
- If the MSG_OOB flag is set by itself, only the last byte of urgent data sent from the remote site is received. To receive multiple bytes of urgent data in the normal data stream, you must set the socket option SO_OOBINLINE, and call recv with the MSG_OOB flag set. recv returns data through the last byte of urgent data. The SO_OOBINLINE socket option is set with either the setsockopt or setsockopt_nw functions. To determine where the last byte of urgent data occurs, use the socket_ioctl() operation SIOCATMARK.

See Nowait Call Errors (page 86) for information on checking errors.

Example

The following programming example calls the recv function. (In the example, rsock is a socket created by a previous call to socket):

```
#include <socket.h>
#include <netdb.h>
...
int status, tosend;
char buffer [8*1024];
...
tosend = sizeof(buffer);
status = recv(rsock, (char *)&buffer[0], tosend, 0);
```

recv64_, recv_nw64_

The recv64_ and recv_nw64_ functions receive data on a connected socket.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

nrcvd = recv64_ (socket, buffer_ptr64, length, flags);

error = recv_nw64_ (socket, buffer_ptr64, length, flags, tag);

       int nrcvd, socket;
       char _ptr64 *buffer_ptr64;
       int length, flags, error;
       long long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

nrcvd := recv64_ (socket, buffer_ptr64, length, flags);

error := recv_nw64_ (socket, buffer_ptr64, length, flags, tag);

       INT(32)     nrcvd,
                   socket;
       STRING .EXT64 buffer_ptr64;
       INT(32)     length,
                   flags,
                   error;
       INT(64)     tag;
```

nrcvd
return value; the number of bytes received by the `recv64_` function. A zero length message indicates end of file (`EOF`).

If the call is not successful, —1 is returned and the external variable *errno* is set as indicated in Errors (page 157).

error
return value; if the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call fails, the external variable *errno* is set as indicated in Errors (page 157).

socket
input value; specifies the socket number for the socket, as returned by the call to `socket` or `socket_nw`.

buffer_ptr64
input and return value; on completion, points to the received data.

length
input value; the size of the buffer pointed to by *buffer_ptr64*.

flags
input value; specifies the kind of data to be read and is one or more of the following:

`MSG_OOB`	Read out-of-band data. This corresponds to the TCP URG flag. The call fails if you select this flag for UDP sockets. This is a constraint imposed by UDP, which does not support out-of-band data.
`MSG_PEEK`	Read the incoming message without removing it from the input queue.
`0`	No flag; read data normally.

tag

input value; the *tag* parameter to be used for the nowait operation initiated by recv_nw64_. For more information, see Asynchrony and Nowaited Operations (page 34).

Errors

If an error occurs, the return value is set to -1, and the external variable *errno* is set to one of the following values:

EHAVEOOB	There is pending out-of-band data. This must be cleared with a call to recv64_ with the MSG_OOB flag set. (This error does not apply to UDP sockets.)
ENOTCONN	The specified socket was not connected.
ESHUTDOWN	The specified socket was shut down.
ETIMEDOUT	The connection timed out.
ECONNRESET	The connection was reset by the remote host.

Usage Guidelines

Use the following guidelines for the recv64_ and recv_nw64_ functions:

- Use recv64_ on a socket created for waited operations. Use recv_nw64_ on a socket created with the socket_nw call for nowait operations. The operation initiated by recv_nw64_ must be completed with a call to the FILE_AWAITIO64_ procedure.
- To determine the number of characters read from recv_nw64_, check the third parameter (the count transferred) returned by the FILE_AWAITIO64_ procedure. For information about the FILE_AWAITIO64_ procedure and its parameters, see *Guardian Procedure Calls Reference Manual*.
- recv64_ and recvfrom64_ might wait indefinitely if the network terminates the connection ungracefully, without returning an error code. This is standard TCP/IP behavior. Avoid the wait by calling recv_nw64_ or recvfrom_nw64_ nowait operations, followed by FILE_AWAITIO64_ call with a timer value of 10 seconds. If the timer expires, call send64_ or sendto64_ from the local host. If the send64_ or sendto64_ call fails, the connection is down.
- The sending side of a connection indicates end-of-file by closing or shutting down its socket. The receiving side recognizes end-of-file when the recv64_ or recvfrom64_ calls have 0 bytes in their *length-of-buffer* field. This is standard practice, not specific to the Guardian socket library implementation.
- If the MSG_OOB flag is set by itself, only the last byte of urgent data sent from the remote site is received. To receive multiple bytes of urgent data in the normal data stream, you must set the socket option SO_OOBINLINE, and call recv64_ with the MSG_OOB flag set. recv64_ call returns data through the last byte of urgent data. The SO_OOBINLINE socket option is set with either the setsockopt or setsockopt_nw functions. To determine where the last byte of urgent data occurs, use the socket_ioctl() operation SIOCATMARK.

For information on checking errors, see Nowait Call Errors (page 86).

Example

The following programming example calls the recv64_ function. (In the example, rsock is a socket created by a previous call to socket):

```
#include <socket.h>
#include <netdb.h>
...
int status, tosend;
char buffer [8*1024];
```

```
...
tosend = sizeof(buffer);
status = recv64_(rsock, (char _ptr64*)&buffer[0], tosend, 0);
```

recvfrom

The `recvfrom` function receives data on an unconnected UDP socket or raw socket created for waited operations.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

nrcvd = recvfrom (socket, buffer_ptr, buffer_length, flags,
                  from_ptr, from_length);

       int nrcvd, socket;
       char *buffer_ptr;
       int buffer_length, flags;
       struct sockaddr * from_ptr;
       int *from_length;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

nrcvd := recvfrom (socket, buffer_ptr, buffer_length, flags,
                   from_ptr, from_length);

        INT(32)       nrcvd,
                      socket;
        STRING .EXT   buffer_ptr;
        INT(32)       buffer_length,
                      flags;
        INT     .EXT  from_ptr(sockaddr_in);
        INT     .EXT  from_length;
```

nrcvd
return value; the number of bytes received. This is the return value.

If the call is not successful, —1 is returned, and the external variable *errno* is set as indicated in Errors (page 159).

socket
input value; specifies the socket number for the socket, as returned by the call to the `socket` function.

buffer_ptr
input value; on return, points to the received data.

buffer_length
input value; the size of the buffer pointed to by *buffer_ptr*.

flags
input value; specifies how the message is read, and is one of the following messages:

`MSG_PEEK`	Read the incoming message without removing it from the queue.
`0`	No flag; read incoming message normally.

from_ptr

input and return value;points, on return, to the remote address and port number (based on the structure `sockaddr_in` or `sockaddr_in6`) from which the data is received.

from_length

input and return value; maintained only for compatibility and should point to a value indicating the size in bytes of the structure (the remote address and port number) pointed to by *from_ptr*.

Errors

If an error occurs, the return value is set to -1 and the external variable *errno* is set to one of the following values:

`EISCONN`	The specified socket was connected.
`ESHUTDOWN`	The specified socket was shut down.
`EINVAL`	An invalid argument was specified.

Usage Guidelines

- This is a waited call; your program pause until the operation completes. Refer to Usage Guidelines (page 155) in the recv, recv_nw function description for more information.
- You can perform a nowait call to receive data on an unconnected UDP socket or raw socket using `recvfrom_nw`, described later in this section.
- Declare the *from_ptr* variable as `struct sockaddr_in6 *` for IPv6 use or as `struct sockaddr_storage *` for protocol-independent use. In C, when you make the call, cast the variable to `sockaddr`. (See the IPv6 example.)

Example

INET: the following programming example calls the `recvfrom` function. In this example, `rsock` is a socket created by a previous call to `socket` and `fhost` is a structure that receives the address of the host from which the data is received. The data is received in `buffer`:

```
#include <socket.h>
#include <in.h>
#include <netdb.h>
...
struct sockaddr_in fhost;
int status, tosend, len;
char buffer [8*1024];
...
tosend = sizeof(buffer);
status = recvfrom(rsock, buffer, tosend,
                  0, (struct sockaddr *)&fhost, &len);
```

INET6: the following programming example calls the `recvfrom` function. In this example, `rsock` is a socket created by a previous call to `socket` and `fhost` is a structure that receives the address of the host from which the data is received. The data is received in `buffer`:

```
#include <socket.h>
#include <in.h>
#include <in6.h>
#include <netdb.h>
...
struct sockaddr_in6 fhost;
int status, tosend, len;
char buffer [8*1024];
...
tosend = sizeof(buffer);
```

```
/* Notice that fhost below is cast to struct sockaddr *
as suggested in the Usage Guidelines */
status = recvfrom(rsock, buffer, tosend,
                  0, (struct sockaddr *)&fhost, &len);
```

recvfrom64_

The `recvfrom64_` function receives data on an unconnected UDP socket or raw socket created for waited operations.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

nrcvd = recvfrom64_ (socket, buffer_ptr64, buffer_length, flags,
                 from_ptr64, from_length64);

       int nrcvd, socket;
       char _ptr64 *buffer_ptr64;
       int buffer_length, flags;
       struct sockaddr _ptr64 * from_ptr64;
       int _ptr64 *from_length64;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

nrcvd := recvfrom64_ (socket, buffer_ptr64, buffer_length, flags,
                  from_ptr64, from_length64);

        INT(32)       nrcvd,
                      socket;
        STRING .EXT64  buffer_ptr64;
        INT(32)       buffer_length,
                      flags;
        INT     .EXT64  from_ptr64(sockaddr_in);
        INT     .EXT64  from_length64;
```

nrcvd
: return value; the number of bytes received.

  If the call is not successful, –1 is returned, and the external variable *errno* is set as shown in Errors (page 161).

socket
: input value; specifies the socket number for the socket, as returned by the call to the `socket` function.

buffer_ptr64
: input value; on return, points to the received data.

buffer_length
: input value; the size of the buffer pointed to by *buffer_ptr64*.

flags

input value; specifies how the message is read, and is one of the following messages:

MSG_PEEK	Read the incoming message without removing it from the queue.
0	No flag; read incoming message normally.

from_ptr64

input and return value; on return, points to the remote address and port number (based on the structure sockaddr_in or sockaddr_in6) from which the data is received.

from_length64

input and return value; maintained only for compatibility and must point to a value indicating the size in bytes of the structure (the remote address and port number) that *from_ptr64* points to.

Errors

If an error occurs, the return value is set to –1, and the external variable *errno* is set to one of the following values:

EISCONN	The specified socket was connected.
ESHUTDOWN	The specified socket was shut down.
EINVAL	An invalid argument was specified.

Usage Guidelines

- This is a waited call; your program pauses until the operation completes. For more information, see Usage Guidelines (page 155) in recv, recv_nw.
- You can perform a nowait call to receive data on an unconnected UDP socket or raw socket using recvfrom_nw64_, described in recvfrom_nw64_ (page 164) call.
- Declare the *from_ptr64* variable as struct sockaddr_in6 * for IPv6 use or as struct sockaddr_storage * for protocol-independent use. In C, when you make the call, cast the variable to sockaddr. (See the IPv6 example.)

Example

INET: the following programming example calls the recvfrom64_ function. In this example, rsock is a socket created by a previous call to socket and fhost is a structure that receives the address of the host from which the data is received. The data is received in buffer:

```
#include <socket.h>
#include <in.h>
#include <netdb.h>
...
struct sockaddr_in fhost;
int status, tosend, len, rsock;
char buffer [8*1024];
...
tosend = sizeof(buffer);
status = recvfrom64_(rsock, (char _ptr64*)&buffer, tosend,
                  0, (struct sockaddr _ptr64*)&fhost, &(int _ptr64*)&len);
```

recvfrom_nw

The recvfrom_nw function receives data on an unconnected UDP socket or raw socket created for nowait operations.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

error = recvfrom_nw (socket, buffer_ptr, buffer_length,
                     flags, r_buffer_ptr, r_buffer_length,
                     tag );

        int error, socket;
        char * buffer_ptr;
        int buffer_length, flags;
        struct sockaddr * r_buffer_ptr;
        int * r_buffer_length;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := recvfrom_nw (socket, buffer_ptr, buffer_length,
                      flags, r_buffer_ptr, r_buffer_length,
                      tag );

        INT(32)      error, socket;
        STRING .EXT  buffer_ptr;
        INT(32)      buffer_length, flags;
        INT .EXT     r_buffer_ptr(sockaddr_in);
        INT(32)      r_buffer_length;
        INT(32)      tag;
```

error
return value; if the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 163).

socket
input value; specifies the socket number for the socket, as returned by the call to `socket_nw`.

buffer_ptr
input and return value; a character pointer to the data returned by the call to `recvfrom_nw`.

buffer_length
input value; the integer length of the data buffer pointed to by *buffer_ptr*.

r_buffer_ptr
input and return value; not used by the `recvfrom_nw` call. Call socket_get_info (page 194) to get the socket address (parameter `sockaddr_buffer`). A dummy parameter must still be passed to satisfy the `recvfrom_nw` call.

r_buffer_length
input and return value; no longer used by the `recvfrom_nw` call to determine the `r_buffer_ptr` size since `r_buffer_ptr` is no longer used; however, `recvfrom_nw` still requires a valid value for this parameter. Call socket_get_info (page 194) to get the socket address structure length (parameter `buflen`).

flags
input value; maintained for compatibility; set to 0.

tag
input value; the *tag* parameter to be used for the nowait operation initiated by `recvfrom_nw`.

Errors

If an error occurs, the return value is set to -1 and the external variable errno is set to one of the following values:

EISCONN	The specified socket was connected.
ESHUTDOWN	The specified socket was shut down.
EINVAL	An invalid argument was specified.

Usage Guidelines

- This is a nowait call; it must be completed with a call to the AWAITIOX procedure. For a waited call, use recvfrom.
- The parameters of the recvfrom_nw function are not compatible with those of the recvfrom function in the 4.3 BSD UNIX operating system.
- The length of the received data is given in the third parameter (count transferred) returned from the AWAITIOX procedure. This length includes the address information given by sizeof(sockaddr_in), sizeof (sockaddr_in6), or sizeof(sockaddr_nv) at the beginning of the buffer.
- For IPv6 use, define the variable *r_buffer_ptr* as a pointer to a structure of type sockaddr_in6.

See Nowait Call Errors (page 86) for information on checking errors.

Examples

INET: the following programming example calls the recvfrom_nw function. In this example, rsock is a socket created by a previous call to socket and fhost is a structure that receives the address of the host from which the data is received. The data is received in buffer:

```
#include <socket.h>
#include <in.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <cextdecs(AWAITIOX, FILE_GETINFO_)>
..
struct sockaddr_in fhost;
int len,rsock;
char buffer [8*1024];
short error, rsock2, rcount;
long tag;
..
error = recvfrom_nw(rsock, buffer, sizeof(buffer), 0,
                    (struct sockaddr *) &fhost, &len, tag);
if error (!= 0)  /* some error checking */
          {
            printf ("recvfrom_nw failed, error %d\n," errno);
            exit (1);
          }
rsock2=(short)rsock; /* AWAITIOX/FILE_GETINFO_ expects a short
                        for socket descriptor */
(void) AWAITIOX (&rsock2,,&rcount,&tag,1l);
(void) FILE_GETINFO_ (rsock2, &error);
if (error != 0)
          {
            printf ("error from AWAITIOX, error %d\n", errno);
            exit (1);
          }
```

```
error = socket_get_info (rsock, (char*) &fhost, len);
if (error != 0)
          {
            printf ("socket_get_info failed, error %d\n", errno);
            exit(1)
          }
```

INET6: the following programming example calls the `recvfrom_nw` function. In this example, `rsock` is a socket created by a previous call to `socket` and `fhost` is a structure that receives the address of the host from which the data is received. The data is received in `buffer`:

```
#include <socket.h>
#include <in.h>
#include <in6.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <cextdecs(AWAITIOX, FILE_GETINFO_)>
..
struct sockaddr_in6 fhost;
int len,rsock;
char buffer [8*1024];
short error, rsock2, rcount;
long tag;
..
error = recvfrom_nw(rsock, buffer, sizeof(buffer), 0,
                    (struct sockaddr *) &fhost, &len, tag);
if error (!= 0)  /* some error checking */
          {
            printf ("recvfrom_nw failed, error %d\n," errno);
            exit (1);
          }
rsock2=(short)rsock; /* AWAITIOX/FILE_GETINFO_ expects a short
                        for socket descriptor */
(void) AWAITIOX (&rsock2,,&rcount,&tag,1l);
(void) FILE_GETINFO_ (rsock2, &error);
if (error != 0)
          {
            printf ("error from AWAITIOX, error %d\n", errno);
            exit (1);
          }
error = socket_get_info (rsock, (char*) &fhost, len);
if (error != 0)
          {
            printf ("socket_get_info failed, error %d\n", errno);
            exit(1)
          }
```

recvfrom_nw64_

The `recvfrom_nw64_` function receives data on an unconnected UDP socket or raw socket created for nowait operations.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

error = recvfrom_nw64_ (socket, buffer_ptr64, buffer_length,
                        flags, addr, r_buffer_length,
                        tag );

        int error, socket;
```

```
        char _ptr64 * buffer_ptr64;
        int buffer_length, r_buffer_length, flags;
        struct sockaddr * addr;
        long long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := recvfrom_nw64_ (socket, buffer_ptr64, buffer_length,
                         flags, r_buffer_ptr, r_buffer_length,
                         tag );

        INT(32)       error, socket;
        STRING .EXT64  buffer_ptr64;
        INT(32)       buffer_length, flags;
        INT .EXT      r_buffer_ptr(sockaddr_in);
        INT(32)       r_buffer_length;
        INT(64)       tag;
```

error
: return value; if the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call fails, the external variable *errno* is set as shown in Errors (page 165).

socket
: input value; specifies the socket number for the socket, as returned by the call to `socket_nw`.

buffer_ptr64
: input and return value; a character pointer to the data returned by the call to `recvfrom_nw64_`.

buffer_length
: input value; the integer length of the data buffer pointed to by *buffer_ptr64*.

r_buffer_ptr
: input and return value; not used by the `recvfrom_nw64_` call. Call `socket_get_info` to get the socket address (parameter `sockaddr_buffer`). A dummy parameter must still be passed to satisfy the `recvfrom_nw64_` call.

r_buffer_length
: input and return value; no longer used by the `recvfrom_nw64_` call to determine the `r_buffer_ptr64` size because `r_buffer_ptr64` is not used; however, `recvfrom_nw64_` still requires a valid value for this parameter. Call socket_get_info (page 194) to get the socket address structure length (parameter `buflen`).

flags
: input value; maintained for compatibility; set to 0.

tag
: input value; the *tag* parameter to be used for the nowait operation initiated by `recvfrom_nw64_`.

Errors

If an error occurs, the return value is set to –1 and the external variable errno is set to one of the following values:

`EISCONN`	The specified socket was connected.
`ESHUTDOWN`	The specified socket was shut down.
`EINVAL`	An invalid argument was specified.

Usage Guidelines

- This is a nowait call; it must be completed with a call to the FILE_AWAITIO64_ procedure. For a waited call, use recvfrom64_.
- The parameters of the recvfrom_nw64_ function are not compatible with those of the recvfrom64_ function in the 4.3 BSD UNIX operating system.
- The length of the received data is specified in the third parameter (count transferred) returned from the FILE_AWAITIO64_ procedure. This length includes the address information given by sizeof(sockaddr_in), sizeof (sockaddr_in6), or sizeof(sockaddr_nv) at the beginning of the buffer.
- For IPv6 use, define the variable *r_buffer_ptr64* as a pointer to a structure of type sockaddr_in6.

For information on checking errors, see Nowait Call Errors (page 86).

Examples

INET: the following programming example calls the recvfrom_nw64_ function. rsock is a socket created by a previous call to socket and fhost is a structure that receives the address of the host from which the data is received. The data is received in buffer:

```
#include <socket.h>
#include <in.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <cextdecs.h>
..
struct sockaddr_in fhost;
int len,rsock, rcount;
char buffer [8*1024];
short error, rsock2;
long long tag;
..
error = recvfrom_nw64_(rsock, (char _ptr64*)&buffer, sizeof(buffer), 0,
                    (struct sockaddr *) &fhost, &len, tag);
if error (!= 0)  /* some error checking */
          {
            printf ("recvfrom_nw64_ failed, error %d\n," errno);
            exit (1);
          }
rsock2=(short)rsock; /* AWAITIOX/FILE_GETINFO_ expects a short
                        for socket descriptor */
(void) FILE_AWAITIO64_ (&rsock2,,&rcount,&tag,1l);
(void) FILE_GETINFO_ (rsock2, &error);
if (error != 0)
          {
            printf ("error from FILE_GETINFO_, error %d\n", errno);
            exit (1);
          }
error = socket_get_info (rsock, (char*) &fhost, len);
if (error != 0)
          {
            printf ("socket_get_info failed, error %d\n", errno);
            exit(1)
          }
```

send

The send function sends data on a connected socket.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

nsent = send (socket, buffer_ptr, buffer_length, flags);

        int nsent, socket;
        char *buffer_ptr;
        int buffer_length, flags;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

nsent := send (socket, buffer_ptr, buffer_length, flags);

        INT(32)      nsent,
                     socket;
        STRING .EXT  buffer_ptr;
        INT(32)      buffer_length,
                     flags;
```

nsent
return value; specifies the number of bytes sent. This is the return value.

If the call is not successful, —1 is returned and the external variable *errno* is set as indicated in Errors (page 167).

socket
input value; specifies the socket number for the socket, as returned by the call to `socket`.

buffer_ptr
input value; points to the data to be sent.

buffer_length
input value; the size of the buffer pointed to by *buffer_ptr*.

flags
input value; specifies the kind of data to be sent, or specifies a routing restriction. *flags* has one of the following values:

`MSG_DONTROUTE`	Send this message only if the destination is located on the local network; do not send the message through a gateway.
`MSG_OOB`	Send the data as out-of-band data. This corresponds to the TCP URG flag.
`0`	Send normal data.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

`EALREADY`	The send buffer is already full.
`EMSGSIZE`	The message was too large to be sent atomically, as required by the socket options.
`ENOTCONN`	The specified socket was not connected.
`ESHUTDOWN`	The specified socket was shut down.
`ETIMEDOUT`	The connection timed out.
`ECONNRESET`	The connection was reset by the remote host.
`EINVAL`	An invalid *flags* value was specified.

EHAVEOOB	There is out-of-band data pending. This must be cleared with a call to recv_nw with the MSG_OOB flag set.
EHAVEOOB	There is out-of-band data pending. This must be cleared with a call to recv_nw with the MSG_OOB flag set.

Usage Guidelines

See Nowait Call Errors (page 86) for information on checking errors.

Example

See UDP Client Program (page 219) for an example that calls send.

send64_

The send64_ function sends data on a connected socket for waited operations.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

nsent = send64_ (socket, buffer_ptr, buffer_length, flags);

        int nsent, socket;
        char _ptr64 *buffer_ptr64;
        int buffer_length, flags;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

nsent := send64_ (socket, buffer_ptr64, buffer_length, flags);

        INT(32)     nsent,
                    socket;
        STRING .EXT64 buffer_ptr64;
        INT(32)     buffer_length,
                    flags;
```

nsent
return value; specifies the number of bytes sent.

If the call is not successful, —1 is returned and the external variable *errno* is set as shown in Errors (page 169).

socket
input value; specifies the socket number for the socket, as returned by the call to socket.

buffer_ptr64
input value; points to the data to be sent.

buffer_length
input value; the size of the buffer pointed to by *buffer_ptr64*.

flags

input value; specifies the type of data to be sent, or specifies a routing restriction. *flags* has one of the following values:

MSG_DONTROUTE	Send the message only if the destination is located on the local network; do not send the message through a gateway.
MSG_OOB	Send the data as out-of-band data. This corresponds to the TCP URG flag.
0	Send the message to the destination. If needed, route the message.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

EALREADY	The send buffer is already full.
EMSGSIZE	The message was too large to be sent atomically, as required by the socket options.
ENOTCONN	The specified socket was not connected.
ESHUTDOWN	The specified socket was shut down.
ETIMEDOUT	The connection timed out.
ECONNRESET	The connection was reset by the remote host.
EINVAL	An invalid *flags* value was specified.
EHAVEOOB	There is out-of-band data pending. This must be cleared with a call to recv_nw64_ with the MSG_OOB flag set.
EHAVEOOB	There is out-of-band data pending. This must be cleared with a call to recv_nw64_ with the MSG_OOB flag set.

Usage Guidelines

For information on checking errors, see Nowait Call Errors (page 86).

Example

The following programming example calls the send64_ function. (In the example, rsock is a socket created by a previous call to socket) .

```
#include <socket.h>
#include <netdb.h>
...
int status, tosend;
char buffer [8*1024];
...
tosend = sizeof(buffer);
status = send64_(rsock, (char _ptr64*)&buffer[0], tosend, 0);
```

send_nw

The send_nw function sends data on a connected socket. send_nw is a nowait operation.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

error = send_nw (socket, nbuffer_ptr, nbuffer_length, flags,
                 tag);

        int error, socket;
        char *nbuffer_ptr;
        int nbuffer_length, flags;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := send_nw (socket, nbuffer_ptr, nbuffer_length,
                  flags, tag);

        INT(32)       error,
                      socket;
        STRING .EXT   nbuffer_ptr;
        INT(32)       nbuffer_length,
                      flags;
        INT(32)       tag;
```

error

return value; if the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 171).

socket

input value; specifies the socket number for the socket, as returned by the call to `socket_nw`.

nbuffer_ptr

input value; points to the element `nb_data[0]` in the following structure:

```
struct  send_nw_str {
        int   nb_sent;
        char  nb_data[1];
};
```

The TAL structure is:

```
struct send_nw_str (*);
   begin
      INT nb_sent;
      STRING nb_data[0:1];
   end;
```

This structure is used by many function calls. Copy the data returned by `nbuffer_ptr` before issuing another function call that uses `nbuffer_ptr`. This structure is provided in the `netdb.h` header file.

nbuffer_length

input value; the size of the buffer pointed to by *nbuffer_ptr*.

flags

input value; specifies the kind of data to be sent, or specifies a routing restriction. *flags* has one of the following values:

`MSG_DONTROUTE`	Send this message only if the destination is located on the local network; do not send the message through a gateway.
`MSG_OOB`	Send the data as out-of-band data. This corresponds to the TCP URG flag.
`0`	Send normal data.

tag

input value; the *tag* parameter to be used for the nowait operation initiated by `send_nw`. (For more information, see Asynchrony and Nowaited Operations (page 34).)

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

EALREADY	The send buffer is already full.
EMSGSIZE	The message was too large to be sent atomically, as required by the socket options.
ENOTCONN	The specified socket was not connected.
ESHUTDOWN	The specified socket was shut down.
ETIMEDOUT	The connection timed out.
ECONNRESET	The connection was reset by the remote host.
EINVAL	An invalid *flags* value was specified.
EHAVEOOB	There is out-of-band data pending. This must be cleared with a call to recv_nw with the MSG_OOB flag set.

Usage Guidelines

- The operation initiated by send_nw must be completed with a call to the AWAITIOX or AWAITIO procedure (although AWAITIOX is recommended).
- To determine the number of bytes that have been transferred as a result of the send_nw function, check nb_sent (the first field of the send_nw_str structure). When the send_nw function completes processing, AWAITIOX returns a pointer to nb_sent as its second parameter and a count of 2 (the length of nb_sent) as its third parameter. This use of the AWAITIOX parameters is nonstandard.

See Nowait Call Errors (page 86) for information on checking errors.

Example

The following programming example calls the send_nw routine and checks for the number of bytes sent:

```
#include <socket.h>
#include <netdb.h>
...
struct send_nw_str *snw;
int cc, count = bp - &buf [0]; errno = 0;
...
for (bp = &buf [0]; count > 0; count -= cc) {
        send_nw (socket, bp, count, 0, 0L);
        AWAITIOX (&ret_fd, (char *)&snw, &cc, &ret_tag, -1L);
        cc = snw->nb_sent;
        if (cc < 0) break;
        bp += cc;
};
```

Before the call to send_nw, the program creates a socket. The socket number is saved in the variable socket. The pointer bp points to the data to be sent. The length of the buffer is count. After the return from AWAITIOX, the program sets cc to the number of bytes in the nb_sent field of the snw structure (based on the send_nw_str structure).

send_nw64_

The send_nw64_ function sends data on a connected socket. send_nw64_ is a nowait operation.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

error = send_nw64_ (socket, nbuffer_ptr64, nbuffer_length, flags,
```

```
                tag);

        int error, socket;
        char _ptr64 *nbuffer_ptr64;
        int nbuffer_length, flags;
        long long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := send_nw64_ (socket, nbuffer_ptr64, nbuffer_length,
                 flags, tag);

        INT(32)      error,
                     socket;
        STRING .EXT64  nbuffer_ptr64;
        INT(32)      nbuffer_length,
                     flags;
        INT(64)      tag;
```

error
return value; if the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call fails, the external variable *errno* is set as shown in Errors (page 173).

socket
input value; specifies the socket number for the socket, as returned by the call to `socket_nw`.

nbuffer_ptr64
input value; points to the element `nb_data[0]` in the following structure:

```
struct  send_nw_str {
        int   nb_sent;
        char  nb_data[1];
};
```

The TAL structure is:

```
struct send_nw_str (*);
   begin
      INT nb_sent;
      STRING nb_data[0:1];
   end;
```

This structure is used by many function calls. Copy the data returned by `nbuffer_ptr64` before issuing another function call that uses `nbuffer_ptr64`. This structure is provided in the `netdb.h` header file.

nbuffer_length
input value; the size of the buffer that *nbuffer_ptr64* points to.

flags
input value; specifies the type of data to be sent, or specifies a routing restriction. *flags* has one of the following values:

`MSG_DONTROUTE`	Send this message only if the destination is located on the local network; do not send the message through a gateway.
`MSG_OOB`	Send the data as out-of-band data. This corresponds to the TCP URG flag.
`0`	Send normal data.

tag
input value; the *tag* parameter to be used for the nowait operation initiated by `send_nw64_`. For more information, see Asynchrony and Nowaited Operations (page 34).

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

EALREADY	The send buffer is already full.
EMSGSIZE	The message was too large to be sent atomically, as required by the socket options.
ENOTCONN	The specified socket was not connected.
ESHUTDOWN	The specified socket was shut down.
ETIMEDOUT	The connection timed out.
ECONNRESET	The connection was reset by the remote host.
EINVAL	An invalid *flags* value was specified.
EHAVEOOB	There is out-of-band data pending. This must be cleared with a call to recv_nw64_ with the MSG_OOB flag set.

Usage Guidelines

- The operation initiated by send_nw64_ must be completed with a call to the FILE_AWAITIO64_ procedure.
- To determine the number of bytes that are transferred as a result of the send_nw64_ function, check nb_sent (the first field of the send_nw_str structure). When the send_nw64_ function completes processing, FILE_AWAITIO64_ returns a pointer to nb_sent as its second parameter and a count of 2 (the length of nb_sent) as its third parameter. This use of the FILE_AWAITIO64_ parameters is nonstandard.

For information on checking errors, see Nowait Call Errors (page 86).

Example

The following programming example calls the send_nw64_ routine and checks for the number of bytes sent:

```
#include <socket.h>
#include <netdb.h>
...
struct send_nw_str *snw;
int cc, count = bp - &buf [0]; errno = 0;
...
for (bp = &buf [0]; count > 0; count -= cc) {
        send_nw64_ (socket, (char _ptr64*)bp, count, 0, 0L);
        FILE_AWAITIO64_ (&ret_fd, (char _ptr64 *)&snw, &cc, &ret_tag, 0D, -1);
        cc = snw->nb_sent;
        if (cc < 0) break;
        bp += cc;
};
```

send_nw2

The send_nw2 function sends data on a connected socket. Unlike the send and send_nw calls, the send_nw2 call does not store the number of bytes sent in the data buffer. Therefore, the send_nw2 call does not require the application to allocate 2 bytes in front of its data buffer to receive the number of bytes sent. Instead, the application should call socket_get_len to obtain the number of bytes sent.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

error := send_nw2 (socket, nbuffer_ptr, nbuffer_length,
                   flags, tag);
```

```
int error, socket;
char *nbuffer_ptr;
int nbuffer_length, flags;
long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := send_nw2 (socket, nbuffer_ptr, nbuffer_length,
                   flags, tag);

          INT(32)      error,
                       socket;
          STRING .EXT  nbuffer_ptr;
          INT(32)      nbuffer_length,
                       flags;
          INT(32)      tag;
```

error
return value; if the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 174).

socket
input value; specifies the socket number for the socket, as returned by the call to `socket_nw`.

nbuffer_ptr
input value; specifies the data to be sent. Call `AWAITIOX` to complete the `send_nw2` call.

nbuffer_length
input value; the size of the buffer pointed to by *nbuffer_ptr*.

flags
input value; specifies the kind of data to be sent, or specifies a routing restriction. *flags* has one of the following values:

`MSG_DONTROUTE`	Send this message only if the destination is located on the local network; do not send the message through a gateway.
`MSG_OOB`	Send the data as out-of-band data. This corresponds to the TCP URG flag.
`0`	Send normal data.

tag
is the *tag* parameter to be used for the nowait operation initiated by `send_nw2`.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

EALREADY	The send buffer is already full.
EMSGSIZE	The message was too large to be sent atomically, as required by the socket options.
ENOTCONN	The specified socket was not connected.
ESHUTDOWN	The specified socket was shut down.
ETIMEDOUT	The connection timed out.
ECONNRESET	The connection was reset by the remote host.
EINVAL	An invalid *flags* value was specified.
EHAVEOOB	There is out-of-band data pending. This must be cleared with a call to `recv_nw` with the MSG_OOB flag set.

Usage Guidelines

- Use send_nw2 on a socket created for nowait operations. The operation initiated by send_nw2 must be completed with a call to the AWAITIOX or AWAITIO procedure (although AWAITIOX is recommended).
- To determine the number of bytes that have been transferred as a result of the send_nw2 function, call the socket_get_len call.
- For the send_nw2 call, complete the request with a call to AWIATIOX before issuing another function call that uses *nbuffer_ptr*.

See Nowait Call Errors (page 86) for information on error checking.

Example

The following programming example calls the send_nw2 routine and checks for the number of bytes sent:

```
#include <socket.h>
#include <netdb.h>
int s;
...
char *snw;
int cc, count = bp - &buf [0]; errno = 0;
...
for (bp = &buf [0]; count > 0; count -= cc) {
        send_nw2 (socket, bp, count, 0, 0L);
        AWAITIOX (&ret_fd, (char *)&snw, &cc, &ret_tag, -1L);
        cc = socket_get_len(s);
        if (cc < 0) break;
        bp += cc;
};
```

Before the call to send_nw2, the program creates a socket. The socket number is saved in the variable socket. The pointer bp points to the data to be sent. The length of the buffer is count. After the return from AWAITIOX, the program sets cc to the number of bytes sent by a call to the socket_get_len function.

send_nw2_64_

The send_nw2_64_ function sends data on a connected socket. Unlike the send, send64_, send_nw, and send_nw64_ calls, the send_nw2_64_ call does not store the number of bytes sent, in the data buffer. Therefore, the send_nw2_64_ call does not require the application to allocate 2 bytes in front of its data buffer to receive the number of bytes sent. Instead, the application must call socket_get_len to obtain the number of bytes sent.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

error := send_nw2_64_ (socket, nbuffer_ptr64, nbuffer_length,
                       flags, tag);

        int error, socket;
        char _ptr64 *nbuffer_ptr64;
        int nbuffer_length, flags;
        long long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC
```

```
error := send_nw2_64_ (socket, nbuffer_ptr64, nbuffer_length,
                       flags, tag);

         INT(32)      error,
                      socket;
         STRING .EXT64 nbuffer_ptr64;
         INT(32)      nbuffer_length,
                      flags;
         INT(64)      tag;
```

error
return value; if the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call fails, the external variable *errno* is set as shown in Errors (page 176).

socket
input value; specifies the socket number for the socket, as returned by the call to socket_nw.

nbuffer_ptr64
input value; specifies the data to be sent. Call FILE_AWAITIO64_ to complete the send_nw2_64_ call.

nbuffer_length
input value; the size of the buffer pointed to by *nbuffer_ptr64*.

flags
input value; specifies the kind of data to be sent, or specifies a routing restriction. *flags* has one of the following values:

MSG_DONTROUTE	Send this message only if the destination is located on the local network; do not send the message through a gateway.
MSG_OOB	Send the data as out-of-band data. This corresponds to the TCP URG flag.
0	Send normal data.

tag
is the *tag* parameter to be used for the nowait operation initiated by send_nw2_64_.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

EALREADY	The send buffer is already full.
EMSGSIZE	The message was too large to be sent atomically, as required by the socket options.
ENOTCONN	The specified socket was not connected.
ESHUTDOWN	The specified socket was shut down.
ETIMEDOUT	The connection timed out.
ECONNRESET	The connection was reset by the remote host.
EINVAL	An invalid *flags* value was specified.
EHAVEOOB	There is out-of-band data pending. This must be cleared with a call to recv_nw64_ with the MSG_OOB flag set.

Usage Guidelines

- Use `send_nw2_64_` on a socket created for nowait operations. The operation initiated by `send_nw2_64_` must be completed with a call to the `FILE_AWAITIO64_` procedure.
- To determine the number of bytes that are transferred as a result of the `send_nw2_64_` function, call the `socket_get_len` call.
- For the `send_nw2_64_` call, complete the request with a call to `AWIATIOX64` before issuing another function call that uses *nbuffer_ptr64*.

For information on error checking, see Nowait Call Errors (page 86).

Example

The following programming example calls the `send_nw2_64_` routine and checks for the number of bytes sent:

```
#include <socket.h>
#include <netdb.h>
int s;
...
char *snw;
int cc, count = bp - &buf [0]; errno = 0;
...
for (bp = &buf [0]; count > 0; count -= cc) {
        send_nw2_64_ (socket, (char _ptr*)bp, count, 0, 0L);
        FILE_AWAITIO64_ (&ret_fd, (char _ptr64*)&snw, &cc, &ret_tag, 0D, -1);
        cc = socket_get_len(s);
        if (cc < 0) break;
        bp += cc;
};
```

sendto

The `sendto` function sends data on an unconnected UDP socket or raw socket for waited operations.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

nsent = sendto (socket, buffer_ptr, buffer_length, flags,
                sockaddr_ptr, sockaddr_length);

        int nsent, socket;
        char *buffer_ptr;
        int buffer_length, flags;
        struct sockaddr *sockaddr_ptr;
        int sockaddr_length;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

nsent := sendto (socket, buffer_ptr, buffer_length, flags,
                 sockaddr_ptr, sockaddr_length);

        INT(32)       socket,
                      buffer_length,
                      flags,
                      sockaddr_length;
        STRING .EXT   buffer_ptr;
        INT    .EXT   sockaddr_ptr(sockaddr);
```

nsent

return value; the number of bytes sent. This is the return value. If this number is less than *length*, the operation should be retried with the remaining data.

If the call is not successful, —1 is returned and the external variable *errno* is set as indicated in Errors (page 178).

socket

input value; specifies the socket number for the socket, as returned by the call to the `socket` function.

buffer_ptr

input value; points to the data to be sent.

buffer_length

input value; the size of the buffer pointed to by *buffer_ptr*.

flags

input value; specifies whether the outgoing data should be sent to the destination if routing is required. This parameter can be one of the following messages:

`MSG_DONTROUTE`	Send this message only if the destination is located on the local network; do not send the message through a gateway.
`0`	No flag; send the message to the destination, even if the message must be routed.

sockaddr_ptr

input value; points to the remote address and port number (based on the structure `sockaddr_in` or `sockaddr_in6`) to which the data is sent.

sockaddr_length

input value; maintained only for compatibility and should be a value indicating the size, in bytes, of the structure (the remote address and port number pointed to by *sockaddr_ptr*.

Errors

If an error occurs, the return value is set to -1 and the external variable *errno* is set to one of the following values:

`EACCES`	Permission denied for broadcast because `SO_BROADCAST` is not set.
`EMSGSIZE`	The message was too large to be sent atomically, as required by the socket options.
`EISCONN`	The specified socket was connected.
`ESHUTDOWN`	The specified socket was shut down.
`ENETUNREACH`	The destination network was unreachable.
`EINVAL`	An invalid argument was specified.

Usage Guidelines

- This is a waited call; your program pauses until the operation is complete.
- Declare the *sockaddr_ptr* variable as `struct sockaddr_in6 *` for IPv6 use or as `struct sockaddr_storage *` for protocol-independent use. In C, when you make the call, cast the variable to `sockaddr`.

Examples

See Client and Server Programs Using UDP (page 219) for examples that call `sendto`.

sendto64_

The `sendto64_` function sends data on an unconnected UDP socket or raw socket for waited operations.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

nsent = sendto64_ (socket, buffer_ptr64, buffer_length, flags, sockaddr_ptr64, sockaddr_len);

        int nsent, socket;
        char _ptr64 *buffer_ptr64;
        int buffer_length, flags, sockaddr_len ;
    struck sockaddr _ptr64 *sockaddr_ptr64;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

nsent := sendto64_ (socket, buffer_ptr64, buffer_length, flags,
                    sockaddr_ptr64, sockaddr_length);

         INT(32)       socket,
                       buffer_length,
                       flags,
                       sockaddr_length;
         STRING .EXT64  buffer_ptr64;
         INT    .EXT64  sockaddr_ptr64(sockaddr);
```

nsent
return value; the number of bytes sent. If this number is less than *length*, the operation must be retried with the remaining data.

If the call is not successful, –1 is returned and the external variable *errno* is set as shown in Errors (page 180).

socket
input value; specifies the socket number for the socket, as returned by the call to `socket`.

buffer_ptr64
input value; points to the data to be sent.

buffer_length
input value; the size of the buffer that *buffer_ptr64* points to.

flags
input value; specifies whether the outgoing data should be sent to the destination if routing is required. This parameter can be one of the following messages:

`MSG_DONTROUTE`	Send this message only if the destination is located on the local network; do not send the message through a gateway.
`0`	No flag; send the message to the destination, even if the message must be routed.

sockaddr_ptr64
input value; contains the remote address and port number to which the data is sent.

sockaddr_len
input value; the size in bytes of *sockaddr_ptr64*.

Errors

If an error occurs, the return value is set to –1, and the external variable *errno* is set to one of the following values:

EACCES	Permission denied for broadcast because SO_BROADCAST is not set.
EMSGSIZE	The message was too large to be sent atomically, as required by the socket options.
EISCONN	The specified socket was connected.
ESHUTDOWN	The specified socket was shut down.
ENETUNREACH	The destination network was unreachable.
EINVAL	An invalid argument was specified.

Usage Guidelines

For information on checking errors, see Nowait Call Errors (page 86).

Example

The following programming example calls the sendto64_ function.

```
#include <socket.h>
#include <netdb.h>
...
int status, tosend, len;
char buffer [8*1024];
...
tosend = sizeof(buffer);
status = sendto64_(channel, (char _ptr64*)&buffer[0], tosend, 0, (struct sockaddr _ptr64*)&remote, len);
```

sendto_nw

The sendto_nw function sends data on an unconnected UDP socket or raw socket created for nowait operations.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

error = sendto_nw (socket, buffer_ptr, buffer_length, flags,
                   sockaddr_ptr, sockaddr_length, tag);

        int error, socket;
        char *buffer_ptr;
        int buffer_length, flags;
        struct sockaddr *sockaddr_ptr;
        intsockaddr_length;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := sendto_nw (socket, buffer_ptr, buffer_length, flags,
                    sockaddr_ptr, sockaddr_length, tag);

        INT(32)         error,
                        socket;
        STRING .EXT     buffer_ptr;
        INT(32)         buffer_length,
                        flags;
        INT     .EXT    sockaddr_ptr(sockaddr);
```

```
INT(32)          sockaddr_length;
INT(32)          tag;
```

error
return value; if the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 181).

socket
input value; specifies the socket number for the socket, as returned by the call to socket_nw.

buffer_ptr
input value; points to the data to be sent.

buffer_length
input value; the size of the buffer pointed to by *buffer_ptr*.

flags
input value; specifies whether the outgoing data should be sent to the destination if routing is required. This parameter can be one of the following messages:

MSG_DONTROUTE	Send this message only if the destination is located on the local network; do not send the message through a gateway.
0	No flag; send the message to the destination, even if the message must be routed.

sockaddr_ptr
input value; points to the remote address and port number to which the data is to be sent. (See the sockaddr_in (page 78), sockaddr_in6 (page 78), and sockaddr_storage (page 79) descriptions.)

sockaddr_length
input value; specifies the length of the sockaddr or sockaddr_in6 structure.

tag
input value; the *tag* parameter to be used for the nowait operation initiated by sendto_nw.

Errors

If an error occurs, the return value is set to -1 and the external variable *errno* is set to one of the following values:

EACCES	Permission denied for broadcast because SO_BROADCAST is not set.
EMSGSIZE	The message was too large to be sent atomically, as required by the socket options.
EISCONN	The specified socket was connected.
ESHUTDOWN	The specified socket was shut down.
ENETUNREACH	The destination network was unreachable.
EINVAL	An invalid argument was specified.

Usage Guidelines

- This is a nowait call; it must be completed with a call to the AWAITIOX procedure. For a waited call, use sendto.
- The parameters of the sendto_nw function are not compatible with those of the sendto function in the 4.3 BSD UNIX operating system.

- To determine the number of bytes transferred as a result of the `sendto_nw` function, use the `socket_get_len` function.
- Declare the *`sockaddr_ptr`* variable as `struct sockaddr_in6 *` for IPv6 use or as `struct sockaddr_storage *` for protocol-independent use. In C, when you make the call, cast the variable to `sockaddr *`.

See Nowait Call Errors (page 86) for information on error checking.

sendto_nw64_

The `sendto_nw64_` function sends data on an unconnected UDP socket or raw socket created for nowait operations.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

error = sendto_nw64_ (socket, buffer_ptr64, buffer_length, flags,
                      sockaddr_ptr64, sockaddr_length, tag);

        int error, socket;
        char _ptr64 *buffer_ptr64;
        int buffer_length, flags;
        struct sockaddr *sockaddr_ptr;
        intsockaddr_length;
        long long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := sendto_nw64_ (socket, buffer_ptr64, buffer_length, flags,
                       sockaddr_ptr, sockaddr_length, tag);

        INT(32)         error,
                        socket;
        STRING .EXT64     buffer_ptr64;
        INT(32)         buffer_length,
                        flags;
        INT     .EXT    sockaddr_ptr(sockaddr);
        INT(32)         sockaddr_length;
        INT(64)         tag;
```

`error`
return value; if the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call fails the external variable *`errno`* is set as shown in Errors (page 183).

`socket`
input value; specifies the socket number for the socket, as returned by the call to `socket_nw`.

`buffer_ptr64`
input value; points to the data to be sent.

`buffer_length`
input value; the size of the buffer that *`buffer_ptr64`* points to.

flags

input value; specifies whether the outgoing data should be sent to the destination if routing is required. This parameter can be one of the following messages:

MSG_DONTROUTE	Send this message only if the destination is located on the local network; do not send the message through a gateway.
0	No flag; send the message to the destination, even if the message must be routed.

sockaddr_ptr

input value; points to the remote address and port number to which the data must be sent. For more information, see sockaddr_in (page 78), sockaddr_in6 (page 78), and sockaddr_storage (page 79).

sockaddr_length

input value; specifies the length of the sockaddr or sockaddr_in6 structure.

tag

input value; the *tag* parameter to be used for the nowait operation initiated by sendto_nw64_.

Errors

If an error occurs, the return value is set to –1, and the external variable *errno* is set to one of the following values:

EACCES	Permission denied for broadcast because SO_BROADCAST is not set.
EMSGSIZE	The message was too large to be sent atomically, as required by the socket options.
EISCONN	The specified socket was connected.
ESHUTDOWN	The specified socket was shut down.
ENETUNREACH	The destination network was unreachable.
EINVAL	An invalid argument was specified.

Usage Guidelines

- This is a nowait call; it must be completed with a call to the FILE_AWAITIO64_ procedure. For a waited call, use sendto64_.
- The parameters of the sendto_nw64_ function are not compatible with those of the sendto64_ function in the 4.3 BSD UNIX operating system.
- To determine the number of bytes transferred as a result of the sendto_nw64_ function, use the socket_get_len function.
- Declare the *sockaddr_ptr 64* variable as struct sockaddr_in6 * for IPv6 use or as struct sockaddr_storage * for protocol-independent use. In C, when you make the call, cast the variable to sockaddr *.

For information on checking errors, see Nowait Call Errors (page 86).

Example

The following programming example calls the sendto_nw64_ function.

```
#include <socket.h>
#include <netdb.h>
...
int socket;
...
struct sockaddr_in fhost;
char *snw;
int cc,len, count = bp - &buf [0]; errno = 0;
...
for (bp = &buf [0]; count > 0; count -= cc) {
```

```
sendto_nw64_(socket,(char _ptr64*) bp, sizeof(bp), 0, (struct sockaddr *)&fhost,len,0L);
FILE_AWAITIO64_(&ret_fd, (char _ptr64*)&snw, &cc, &ret_tag, 0D,-1);
cc = socket_get_len(socket);
if (cc < 0) break;
bp += cc;
};
```

setsockopt, setsockopt_nw

The `setsockopt` and `setsockopt_nw` functions set the socket options for a socket.

NOTE: In CIP, certain setsockopt and setsockopt_nw operations are not supported or may have different defaults or different behavior. See the *Cluster I/O Protocols (CIP) Configuration and Management Manual* for details.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <netdb.h>

error = setsockopt (socket, level, optname, optval_ptr,
                    optlen);

error = setsockopt_nw (socket, level, optname, optval_ptr,
                       optlen, tag);

        int error, socket, level, optname;
        char *optval_ptr;
        int optlen;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := setsockopt (socket, level, optname, optval_ptr,
                     optlen);

error := setsockopt_nw (socket, level, optname, optval_ptr,
                        optlen, tag);

        INT(32)      error,
                     socket,
                     level,
                     optname;
        STRING .EXT  optval_ptr;
        INT(32)      optlen;
        INT(32)      tag;
```

error
: return value; f the call is successful, a zero is returned. If the call is not successful, —1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 188).

socket
: input value; specifies the socket number for the socket, as returned by the call to `socket` or `socket_nw`.

level

input value; the socket level at which the socket option is being managed. The possible values are:

SOL_SOCKET	Socket-level option.
IPPROTO_TCP	TCP-level option.
IPPROTO_IP	IP-level option for INET sockets.
IPPROTO_IPV6	IP-level option for INET6 sockets.
IPPROTO_ICMP	ICMP-level option.
IPPROTO_RAW	Raw-socket level option.
user-protocol	Option for a user-defined protocol above IP, such as PUP.

user-protocol can be any protocol number other than the numbers for TCP, UDP, IP, ICMP, and RAW. Appendix A (page 241), lists the protocol numbers.

optname

input value; the socket option name.

When *level* is SOL_SOCKET, the possible values are:

SO_BROADCAST	Broadcast messages when data is sent. For UDP sockets, see Usage Guidelines (page 188)
SO_ERROR	Get the error status and clear the socket error. This option applies only to the getsockopt function.
SO_TYPE	Get the socket type. This option applies only to the getsockopt and getsockopt_nw functions. The possible values are: SOCK_STREAMStream socket. SOCK_DGRAM Datagram socket. SOCK_RAW Raw socket.
SO_DONTROUTE	Do not route messages.
SO_REUSEADDR	Allow reuse of local port addresses in a bind operation.
SO_LINGER	Cause connections to close gracefully, and wait for data transfer to complete. This option is provided for compatibility only. All TCP/IP connections close gracefully.
SO_KEEPALIVE	Keep connections alive during inactivity by sending "keep-alive" messages. A keep-alive message is a probe segment that causes the receiver to return an acknowledgment segment, confirming that the connection is still alive. For a detailed description of this mechanism, see RFC 1122.
SO_OOBINLINE	Keep out-of-band data in with normal data. If out-of-band data is kept with the normal data, the application must discard normal data until the out-of-band data is read.
SO_SNDBUF	Set the size of the send window. The SO_RCVBUF and SO_SNDBUF options are used as hints for determining how much space to allocate in the underlying network I/O buffers. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. See Usage Guidelines (page 188).
SO_RCVBUF	Set the size of the receive window. The SO_RCVBUF and SO_SNDBUF options are used as hints for determining how much space to allocate in the underlying network I/O buffers. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. See Usage Guidelines (page 188) and Considerations for a Server Posting Receives (page 35).
SO_REUSEPORT	Allow local address and port reuse for UDP sockets receiving multicast datagrams. See "Receiving IPv4 Multicast Datagrams" (page 45)

When *level* is IPPROTO_IP, the value is:

IP_OPTIONS	Set IP options for each outgoing packet. *optval_ptr* is a pointer to a list of IP options and values whose format is as defined in RFC 791.
IP_MULTICAST_IF	Set the multicast interface IP address (that is, subnet IP address) to which the multicast output is destined. A default interface is chosen if this option is not set or is set to INADDR_ANY.
IP_MULTICAST_TTL	Set Time-To-Live for multicast datagram. Default TTL is 1.
IP_MULTICAST_LOOP	Enable(1) or disable(0) loopback of messages sent to multicast groups. Default is loopback-enabled.
IP_ADD_MEMBERSHIP	Add a multicast group to the socket. If the associated interface IP address is set to INADDR_ANY or in6addr_any, a default interface is chosen.
IP_DROP_MEMBERSHIP	Delete a multicast group from the socket.

When *level* is IPPROTO_IPV6, the value is:

IPV6_MULTICAST_IF	Set the multicast interface IP address (that is, subnet IP address) to which the multicast output is destined. A default interface is chosen if this option is not set or is set to in6addr_any for IPv6.
IPV6_MULTICAST_HOPS	Set Time-To-Live for multicast datagram. Default TTL is 1.
IPV6_MULTICAST_LOOP	Enable(1) or disable(0) loopback of messages sent to multicast groups. Default is loopback-enabled.
IPV6_JOIN_GROUP	Add a multicast group to the socket. If the associated interface IP address is set to INADDR_ANY or in6addr_any, a default interface is chosen.
IPV6_LEAVE_GROUP	Delete a multicast group from the socket.
IPV6_V6ONLY	AF_INET6 sockets are restricted to IPv6–only communication.

When *level* is IPPROTO_TCP, you should include the tcp.h file. The value is:

TCP_NODELAY	Do not buffer data packets before sending them. TCP_NODELAY is recommended where per-character buffering and acknowledgment is inefficient; for example, in a non-character-based application such as a terminal emulator client sending mouse and window movement information to a terminal server.
TCP_SACKENA	Enables TCP selective acknowledgements.
TCP_MINRXMT	Sets the minimum time for TCP retransmission timeout. The default is 1 second. The range is 500 milliseconds to 30 seconds.
TCP_MAXRXMT	Sets the maximum time for a TCP retransmission timeout. The default is 64 seconds. The range is 500 milliseconds to 20 minutes.
TCP_RXMTCNT	Sets the maximum number of continuous retransmissions prior to dropping a TCP connection. The default is 12. The range is 1 to 12.
TCP_TOTRXMTVAL	Sets the maximum continuous time spent retransmitting without receiving an acknowledgement from the other endpoint. The default is 12 minutes. The range is 500 milliseconds to 4 hours.

When *level* is a user-defined protocol above IP, the possible values are defined by the protocol.

optval_ptr

input value; points to the value of the socket option, specified by *optname*, which is passed to the level specified in *level*. Table 14 and Table 15 list the type and length of the value of each socket option. Boolean-type values are integers, where 0 indicates false and 1 indicates true.

optlen

input value; the length, in bytes, of the list pointed to by *optval_ptr*. If too small, the error EINVAL is returned. (See Errors (page 188).)

tag

input value; the *tag* parameter to be used for the nowait operation initiated by setsockopt_nw.

Table 14 Types and Lengths of Socket Option Values

Socket Option	Type
SO_BROADCAST	Integer (Boolean)
SO_ERROR	Integer
SO_TYPE	Integer
SO_DONTROUTE	Integer (Boolean)
SO_REUSEADDR	Integer (Boolean)

SO_LINGER	Struct linger {
	short l_onoff; /*boolean*/
	short l_linger; /*time*/
	};

SO_KEEPALIVE	Integer (Boolean)
SO_OOBINLINE	Integer (Boolean)
SO_SNDBUF	Integer
SO_RCVBUF	Integer
IP_OPTIONS	Integer
TCP_NODELAY	Integer (Boolean)
TCP_SACKENA	Integer (Boolean)
TCP_MINRXMT	Integer
TCP_MAXRXMT	Integer
TCP_RXMTCNT	Integer
TCP_TOTRXMTVAL	Integer
IP_MULTICAST_IF	struct in_addr
IPV6_MULTICAST_IF	integer
IP_MULTICAST_TTL	char
IPV6_MULTICAST_HOPS	integer
IP_MULTICAST_LOOP	char
IPV6_MULTICAST_LOOP	integer
IP_ADD_MEMBERSHIP	struct ip_mreq
IPV6_JOIN_GROUP	struct ipv6_mreq
IP_DROP_MEMBERSHIP	struct ip_mreq
IPV6_LEAVE_GROUP	struct ipv6_mreq

Note: For Boolean options, the option value should be set to TRUE or a nonzero value to enable the option; the option value should be set to 0 (zero) or FALSE to disable the option.

Errors

If an error occurs, the external variable *errno* is set to one the following values:

ENOPROTOOPT	The specified option is unknown to the protocol.
EINVAL	An invalid argument was specified.

Usage Guidelines

- Use setsockopt on a socket created for waited operations, or setsockopt_nw on a socket created for nowait operations. The operation initiated by P/setsockopt_nw must be completed with a call to the AWAITIOX procedure.
- When a packet is sent from an application to a broadcast address, the packet is received by the local host unless a duplicate packet is also sent to the loopback address.
- When you call the setsockopt or setsockopt_nw function for UDP sockets, the SO_BROADCAST option must be specified if you want to send a broadcast packet.
- If packets larger than the default values need to be sent or received, specify the appropriate size in the SO_SNDBUF and SO_RCVBUF socket options, respectively. The following table summarizes the default values for each subsystem:

	NonStop TCP/IP	Parallel Library TCP/IP	NonStop TCP/IPv6
SO_SNDBUF TCP Default	8,192 bytes	8,192 bytes	61,440 bytes
SO_SNDBUF UDP Default	9,216 bytes	9,216 bytes	9,216 bytes
SO_RCVBUF UDP Default	20,800 bytes	41,600 bytes	42,080 bytes

  The maximum values for these two options for NonStop TCP/IP and Parallel Library TCP/IP is 262,144 bytes. The maximum value for these two options for NonStop TCP/IPv6 is 1,048,576 bytes. (Anything over 32,767 must be passed using the wide model.) An interprocess transfer is restricted to 32,000 bytes for NonStop TCP/IP and to 57,344 bytes for Parallel Library TCP/IP and NonStop TCP/IPv6. Refer to the discussion of WRITEREAD[X] in the *Guardian Procedure Calls Reference Manual* for more information.
- Applications can use the SETSOCKOPT call options to alter, on an individual TCP socket basis, the TCP retransmission timer variables.
- All time values used for the socket library calls are in 500 millisecond ticks.
- If the TCP maximum retransmission count (TCP_RXMTCNT) multiplied by the TCP maximum retransmission timeout (TCP_MAXRXMT) is lower than the total maximum TCP retransmission duration, the TCP connection is dropped sooner than the duration value.
- The TCP_MAXRXMT value should be set to be greater (or at least equal too) the TCP_MINRXMT value.
- Socket options for incoming connections that are accepted with a call to accept_nw2 should not be set until the accept_nw2 call completes. Any socket options that are set prior to the call to accept_nw2 are lost.

See Nowait Call Errors (page 86) for information on error checking. See also Dropping Membership in a Multicast Group (page 61).

Examples

See UDP Client Program (page 219) for examples that call the setsockopt routine.

shutdown, shutdown_nw

The shutdown and shutdown_nw functions shut down data transfer, partially or completely, on an actively connected TCP socket.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

error = shutdown (socket, how);

error = shutdown_nw (socket, how, tag);

        int error, socket, how;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := shutdown (socket, how);

error := shutdown_nw (socket, how, tag);

        INT(32) error,
                socket,
                how;
        INT(32) tag;
```

error
: return value; if the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 189).

socket
: input value; specifies the socket number for the socket, as returned by the call to socket or socket_nw.

how
: input value; specifies what kind of operations on the socket are to be shut down. It must be one of the following values:

0	Disallow further reads (calls to recv and recv_nw).
1	Disallow further writes (calls to send, send_nw, and send_nw2)
2	Disallow both reads and writes.

tag
: is the *tag* parameter to be used for the nowait operation initiated by shutdown_nw.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

EINVAL	An invalid value was passed for the *how* parameter.
ENOTCONN	The specified socket was not connected or already shut down.

Usage Guidelines

- Use `shutdown` on a socket created for waited operations, or `shutdown_nw` on a socket created for nowait operations. The operation initiated by `shutdown_nw` must be completed with a call to the `AWAITIOX` procedure.
- Because the `shutdown` function shuts down an active connection, it has no meaning for the UDP or IP protocols.
- After a socket is shut down, there is a delay before the port can be reused. This delay occurs so that any stray packets can be flushed from the network. The length of the delay varies, based on the average round-trip time for packets in the network.
- The `shutdown` and `shutdown_nw` functions do not destroy the socket. To destroy a socket, call the `FILE_CLOSE` procedure to destroy it.

See Nowait Call Errors (page 86) for information on error checking.

Example

The following example calls the `shutdown` function. (Data transfer on socket `s1` is shutdown; no further reads or writes are allowed.):

```
#include <socket.h>
#include <netdb.h>
...
\* Code to create socket s1, connect socket to server,
 * and transfer data appears here.
 *\
...
\* When finished transferring data, execute the following
 * code.
 *\
if (shutdown (s1, 2) < 0)
      perror ("Shutdown failed.");
```

sock_close_reuse_nw

The `sock_close_reuse_nw` function is for use by servers that accept using the functions `accept_nw` and `accept_nw2`. It replaces the `close()` function for an existing socket, marks the socket for reuse and eliminates the need for a new socket to be created for the `accept_nw2()` function call. The intention of this function is to improve performance by eliminating socket close and open processing.

The `sock_close_reuse_nw` function is intended only for non fault-tolerant sockets (`SOCK_STREAM_NONFT`). If the `sock_close_reuse_nw` function is used on a fault-tolerant socket (`SOCK_STREAM`), the socket is closed and error EINVAL is returned to the application.

C Synopsis

```
#include <netdb.h>
error = sock_close_reuse_nw(socket, tag);

        int error, socket;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := sock_close_reuse_nw(socket, tag);

         INT(32) error,socket;
         INT(32) tag;
```

error
return value; if the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 191).

socket
input value; specifies the socket number for the socket, as returned by the call to socket or `socket_nw`.

tag
input value; the tag parameter to be used for the nowait operation.

Errors

EINVAL: An invalid argument was specified.

ENOTCONN: The specified socket is not connected.

Usage Guidelines

- This is a nowait call; it must be completed with a call to the AWAITIOX procedure.
- See Nowait Call Errors (page 86) for information on error checking.
- The application needs to keep a list of sockets which have been marked for reuse by this call. When a socket would normally be closed, the `close()` call is replaced with the `sock_close_reuse_nw()` call and the socket added to the list. If any sockets exist on this list when an `accept_nw()` call completes, the `socket()` call can be omitted and the `accept_nw2()` is passed the socket found on the list. The socket is then removed from the list.
- You must set the socket type as `sock_stream_nonft` instead of instead of the standard `sock_stream` to use this call.

Table 15 Comparison of Socket Calls With and Without `sock_close_reuse_nw`

With sock_close_reuse_nw()	Without sock_close_reuse_nw()
accept_nw	accept_nw
socket = socket_nw	socket = socket_nw
accept_nw2(socket)	accept_nw2(socket)
...	...
sock_close_reuse_nw(socket)	close(socket)
accept_nw	accept_nw
accept_nw2(socket)	socket = socket_nw
...	accept_nw2(socket)
sock_close_reuse_nw(socket)	...
	close(socket)

- When an application tries to mark a fault-tolerant socket (`SOCK_STREAM`) for reuse, error EINVAL is returned. If the application ignores this error and continues to use the socket on it's subsequent `accept_nw2()` function, error EWRONGID is returned.

socket, socket_nw

The `socket` function creates a socket for waited operations; the `socket_nw` function creates a socket for nowait operations.

C Synopsis

```
#include <socket.h>
#include <netdb.h>

socket_file_number = socket (address_family, socket_type,
                             protocol);

socket_file_number = socket_nw (address_family,socket_type,
                                protocol, flags, sync);

      int socket_file_number, address_family, socket_type,
          protocol, flags, sync;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

socket_file_number := socket (address_family, socket_type,
                              protocol);

socket_file_number := socket_nw (address_family,socket_type,
                                 protocol, flags, sync);

      INT  socket_file_number,
           address_family,
           socket_type,
           protocol,
           flags,
           sync;
```

socket_file_number
return value; the socket number for the newly created socket. If the call is not successful –1 is returned, and the external variable *errno* is set as indicated in Errors (page 193).

address_family
input value; specifies the address format. The value given for this parameter must be AF_INET for NonStop TCP/IP and Parallel Library TCP/IP but can be either AF_INET or AF_INET6 for NonStop TCP/IP.

socket_type
input value; specifies the semantics of communication. It must be one of the following values:

SOCK_STREAM	Create a TCP socket.
SOCK_STREAM_NONFT	Create a non-fault-tolerant socket.
SOCK_DGRAM	Create a UDP socket.
SOCK_RAW	Create a raw socket for access to the raw IP protocol level. To create a raw socket, the process access ID of the requesting application must be in the SUPER group (user ID 255,*nnn*).

protocol
input value; the specific IP number. This parameter must be specified if *socket_type* is SOCK_RAW; it is ignored if *socket_type* is SOCK_STREAM or SOCK_DGRAM.

If *socket_type* is SOCK_RAW, the value of *protocol* cannot be the number assigned to ICMP (1), TCP (6), or UDP (17). The application must provide support for the specified protocol.

flags
input value; specified in the format of the *flags* parameter for the deprecated OPEN_() procedure, as described in the *Guardian Procedure Errors and Messages Manual*.

The following considerations apply to this parameter:

- The function `socket_nw()` internally maps the old `FLAGS` parameter to the corresponding parameters for the `FILE_OPEN_()`.
- The *`flags`* parameter is not used for the `socket` function (waited operations). For the `socket_nw` function, flags.< bit 8> = 1 indicates a nowaited file open and flags.< bits 12:15> indicates the maximum number of outstanding nowaited I/Os allowed (nowait depth).

`sync`

input value; not supported for Guardian sockets. It must always be set to zero.

Errors

If an error occurs, the return value is set to -1 and the external variable *`errno`* is set to one of the following values:

`EAFNOSUPPORT`	The address family specified in *`address_family`* is not supported.
`ESOCKTNOSUPPORT`	The socket type specified in *`socket_type`* is not supported.
`EPROTONOSUPPORT`	The protocol specified was not in the range 0 to 255, or was the value reserved for TCP, UDP, or ICMP.

Usage Guidelines

- The `socket` or `socket_nw` function opens the NonStop TCP/IP or TCP6SAM process by name; therefore, the function must know the name of this process. If your program calls the `socket_set_inet_name` function before calling the `socket` or `socket_nw` function, the socket library opens the process you specify.

  If your program does not call `socket_set_inet_name`, the socket library opens the process with the name defined for `=TCPIP^PROCESS^NAME`. If a defined name does not exist, the socket library uses the process name `$ZTC0`. For more information on `=TCPIP^PROCESS^NAME`, see Using the DEFINE Command (page 29).
- When a nowaited socket open (flags.< bit8> = 1) is specified:
  - The socket_nw() must be completed by calling AWAITIOX().
  - Tag returned is -30D.
  - SETMODE 30 must be called to allow I/O operations to complete in any order.
- To allow nowaited I/O operations, a socket must have nowait depth > 0 (flags.< bit 12:15>). The nowait versions (_nw) of the socket routines must be used for subsequent operations on the socket.
- For nowait operations on a socket, set a nowait depth >= 2 to allow pending simultaneous reads and writes.

See Nowait Call Errors (page 86) for information on error checking.

Example

See accept_nw (page 91) for an example that uses a call to `socket_nw`.

socket_backup

The `socket_backup` function returns data to the backup process of a NonStop process pair, after the primary process has checkpointed the data using the `socket_get_open_info` function. This function is designed to allow applications to establish a backup open to a NonStop TCP/IP, TCPSAM, or TCP6SAM process.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <if.h>
#include <netdb.h>

error = socket_backup(*message, *brother_phandle);

        int error;
        struct open_info_message *message;
        char *brother_phandle;
```

error
: return value; if the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 194).

message
: input value; refer to the FILE_OPEN_ procedure call in the *Guardian Procedure Errors and Messages Manual* for a description of this field. The open_info_message structure is shown in Chapter 4 (page 81).

brother_phandle
: input value; refer to the FILE_OPEN_ procedure call in the *Guardian Procedure Calls Reference Manual* for a description of this field.

Errors

File-system errors as defined in <errno.h> are returned by this call. For a description of the file-system error returned, type (from the TACL prompt):

```
> ERROR error-num
```

where *error-num* is the error number returned in *errno*.

Usage Guideline

The user need only checkpoint the open information for the listening socket, as all open sockets are closed as a result of the backup application takeover and an ECONNRESET returned to all operations on these sockets. The application is then responsible for end-to-end re-synchronization of the data stream. Upon takeover, the backup process is therefore only required to post a new listen on the existing (checkpointed) socket by issuing a call to accept_nw().

The *message* is the information that was checkpointed as a result of the primary process calling socket_get_open_info(). The brother_phandle is the phandle of the primary application process and can be obtained from a call to PROCESS_GETPAIRINFO. Refer to the FILE_OPEN_() procedure call in the *Guardian Procedure Calls Reference Manual* for more information on handling backup opens.

socket_get_info

The socket_get_info function returns the sockaddr data structure and the sockaddr length received after a recvfrom_nw call.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <if.h>
#include <netdb.h>
```

```
error = socket_get_info(socket, sockaddr_buffer, buflen);

        int  error, socket;
        char *sockaddr_buffer;
        int buflen;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := socket_get_info(socket, sockaddr_buffer, buflen);

        INT(32)     error, socket;
        STRING .EXT sockaddr_buffer;
        INT(32)     buflen;
```

error
: return value; if the call is successful, the size of the `sockaddr` data structure is returned. If the call is not successful, –1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 195).

socket
: input value; the socket specified in the prior `recvfrom_nw` call.

sockaddr_buffer
: input and return value; a character pointer to the `sockaddr_in` or `sockaddr_nv` data structure returned by the call.

buflen
: input value; the size of *sockaddr_in_buffer* or *sockaddr_nv_buffer* in bytes. Maximum value is 80 bytes.

Examples

See Examples (page 163) for recvfrom_nw.

Errors

If an error occurs, the variable *error* is set to one of the following values:

EMSGSIZE	The message was too large to be sent atomically, as required by the socket options.
ENOTCONN	The specified socket was not connected.
ESHUTDOWN	The specified socket was shut down.
EINVAL	An invalid argument was specified.

Usage Guideline

Use `socket_get_info` to retrieve the `sockaddr_in` or `sockaddr_nv` data structure and the length of the *sockaddr_in_buffer* or *sockaddr_nv_buffer*, after a call to `recvfrom_nw` and `AWAITIOX` and before a subsequent `AWAITIOX` call.

socket_get_len

The `socket_get_len` function returns the number of bytes sent following a `sendto_nw` or `send_nw2` call.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
```

```
bytes_sent = socket_get_len(socket);

        int bytes_sent, socket;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKPROC

bytes_sent := socket_get_len(socket);

        INT(32)     bytes_sent,
                    socket;
```

bytes_sent
return value; the number of bytes sent from a `sendto_nw` call or a `send_nw2` call.

socket
input value; the socket specified in the prior `sendto_nw` or `send_nw2` call.

Errors

There are no errors returned by this call.

Usage Guideline

Use `socket_get_len` after a call to `AWAITIOX` and before a subsequent call to `AWAITIOX`.

socket_get_open_info

The `socket_get_open_info` function is used by the primary process in a NonStop TCP/IP process pair to get parameters following a `socket` or `socket_nw` call.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h>
#include <if.h>
#include <netdb.h>

error = socket_get_open_info(*message);

        int error;
        struct open_info_message *message;
```

error
return value; if the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 196).

message
input and return value; refer to the `FILE_OPEN_` procedure call in the *Guardian Procedure Calls Reference Manual* for a description of this field. The `open_info_message` structure is shown open_info_message (page 72).

Errors

File-system errors as defined in `<errno.h>` are returned by this call. For a description of the file-system error returned, type (from the TACL prompt):

```
> ERROR error-num
```

where *error-num* is the error number returned in *errno*.

Usage Guidelines

- Use `socket_get_open_info` after creating a socket using the `socket` or `socket_nw` functions. Then, immediately checkpoint the data.
- Use `socket_get_open_info` to checkpoint state information to a backup process after a call to `AWAITIOX` and before subsequent `AWAITIOX` calls.
- The user application must fill in the *`filenum`*, *`flags`* and *`sync`* variables in the open_info_message structure before calling this function. *`Flags`* and *`sync`* must have the same values that were used in the call to `socket()/socket_nw()` that resulted in the opening of the socket identified by *`filenum`*. Immediately after the call to `socket_get_open_info()`, the user application must checkpoint the information by whatever means is being employed (passive or active) to its backup process.

socket_ioctl, socket_ioctl_nw

The `socket_ioctl` and `socket_ioctl_nw` functions perform a control operation on a socket.

NOTE: In CIP, certain socket_ioctl and socket_ioctl_nw operations are not supported, may have different defaults, or have different behavior. See the *Cluster I/O Protocols (CIP) Configuration and Management Manual* for details.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <in6.h> /* for IPv6 use */
#include <if.h>
#include <route.h>
#include <mbuf.h>
#include <ioctl.h>
#include <netdb.h>

error = socket_ioctl (socket, command, arg_ptr);

error = socket_ioctl_nw (socket, command, arg_ptr, tag);

        int error, socket, command;
        long tag;
        char *arg_ptr;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := socket_ioctl (socket, command, arg_ptr);

error := socket_ioctl_nw (socket, command, arg_ptr, tag);

        INT(32)     error,
                    socket,
                    command;
        STRING .EXT arg_ptr;
        INT(32)     tag;
```

error

return value; f the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 198).

socket
input value; specifies the socket number for the socket, as returned by the call to `socket` or `socket_nw`.

command
input value; specifies the operation to be performed on the socket. Supported operations are listed in Table 16 (page 199).

arg_ptr
input value; points to the argument for the operation. The pointer type is dependent on the value of *command*. See Table 16 (page 199) for a list of the pointer types.

tag
input value; the *tag* parameter to be used for the nowait operation initiated by `socket_ioctl_nw`.

Errors

If an error occurs, the external variable *errno* is set to one of the errors listed in Appendix B (page 243); the possible errors depend on the value of *command*. Most of the commands return the following errors:

`EINVAL`	An invalid argument was specified.
`EPERM`	The specified operation cannot be performed by a nonprivileged user.

Usage Guidelines

- Use `socket_ioctl` on a socket created for waited operations, and `socket_ioctl_nw` on a socket created for nowait operations. The operation initiated by `socket_ioctl_nw` must be completed with a call to the `AWAITIOX` procedure.
- In general, `socket_ioctl` and `socket_ioctl_nw` control operations are provided for compatibility only. To alter network parameters or to determine their values, it is recommended that you use the Distributed Systems Management (DSM) ADD, ALTER, DELETE, and INFO commands. The interactive versions of these commands are described in the *TCP/IPv6 Configuration and Management Manual*.
- The following commands (listed in Table 16 (page 199)) can be performed only by applications whose process access ID is in the SUPER group (user ID 255,*nnn*):

`SIOCSIFBRDADDR`	`SIOCSIFADDR**`
`SIOCSIFDSTADDR`	`SIOCSIFFLAGS`
`SIOCADDRT**`	`SIOCDELRT**`
`SIOCSIFNETMASK**`	`SIOCSIFMETRIC`
`SIOCSARP`	`SIOCDARP`

The commands marked with double asterisks (**) can be accessed using the DSM commands as follows:

`SIOCSIFADDR`	Can be accessed through the ZIP-ADDR attribute of the ZCOM-OBJ-SUBNET type by using the programmatic ALTER command (ZCOM-CMD-ALTER).
`SIOCADDRT`	Can be accessed through the ADD ROUTE command (ZCOM-CMD-ADD for the ZCOM-OBJ-ROUTE object type).
`SIOCDELRT`	Can be accessed through the DELETE ROUTE command (ZCOM-CMD-DELETE for the ZCOM-OB-ROUTE object type).
`SIOCSIFNETMASK`	Can be accessed through the ZSUBNET-MASK attribute of the ZCOM-OBJ-SUBNET type by using the programmatic ALTER command (ZCOM-CMD-ALTER).

- The FIONBIO command is not supported. If this command is selected, the EINVAL error is returned.
- If you select FIONREAD for UDP sockets, the number of characters returned is greater than the number of characters received as a result of a call to the recv or recvfrom functions; the increase in characters is equal to sizeof (struct sockaddr_in). The additional characters are returned because the network keeps the sender's socket address at the beginning of the data until the application requests the data.
- UDP does not support out-of-band data. Use of the command argument SIOCATMARK is meaningless for UDP, although specifying SIOCATMARK does not cause the call to fail.
- The SIOCSIFFLAGS function is now disabled. The call completes successfully but no flags are changed.
- For SIOCGIFCONF, the data-buffer pointer (ifc_buf) must point to the first byte immediately following the ifconf structure, because the Parallel Library TCP/IP, NonStop TCP/IPv6, and NonStop TCP/IP architectures allow only a single buffer to be passed.
- For SIOCGIFNUM, aliases are not included in the count.

See Nowait Call Errors (page 86) for information on error checking.

Socket I/O Control Operations

Table 16 gives the I/O control operations that can be specified in *command*, the corresponding pointer types for *arg_ptr*, and descriptions of the commands. The definitions of the structures pointed to by *arg_ptr* are provided in Chapter 3 (page 62).

Table 16 Socket I/O Control Operations

Command	Pointer Type for *arg*	Description
FIONREAD	int *	Get the number of bytes waiting to be read.
SIOCSIFADDR	struct ifreq *	Set the interface address. Returns the error [EOPNOTSUPP].
SIOCGIFADDR	struct ifreq *	Get the interface address.
SIOCGIFCONF	struct ifconf *	Get the interface configuration list. See Usage Guidelines (page 198).
SIOCGIFNUM	int *	Get the number of interfaces that have been configured. See Usage Guidelines (page 198).
SIOCSIFDSTADDR	struct ifreq *	Set the destination address on a point-to-point interface. Returns the error [EOPNOTSUPP].
SIOCGIFDSTADDR	struct ifreq *	Get the destination address on a point-to-point interface.
SIOCSIFFLAGS	struct ifreq *	Set the interface flags. Returns the error [EOPNOTSUPP].
SIOCGIFFLAGS	struct ifreq *	Get the interface flags.
SIOCADDRT	struct rtentry *	Add a specific route.
SIOCDELRT	struct rtentry *	Delete a specific route.
SIOCATMARK	int *	Check for pending urgent data. If a nonzero value is returned, urgent data is pending.

Table 16 Socket I/O Control Operations *(continued)*

Command	Pointer Type for *arg*	Description
SIOCSIFBRDADDR	`struct ifreq *`	Set the broadcast address associated with a subnet device. Returns the error [EOPNOTSUPP].
SIOCGIFBRDADDR	`struct ifreq *`	Get the broadcast address associated with a subnet device.
SIOCSIFNETMASK	`struct ifreq *`	Set the network address mask. `SIOCSIFNETMASK` specifies which portion of the IP host ID and IP network number should be masked to define a subnet. Returns the error [EOPNOTSUPP].
SIOCGIFNETMASK	`struct ifreq *`	Get the network address mask.
SIOCSARP	`struct arpreq *`	Set an ARP protocol (IP address/hardware address pair) address entry in the translation table. This address is distinct from the ARP hardware address.
SIOCGARP	`struct arpreq *`	Get an ARP protocol address entry (hardware address) from the translation table.
SIOCDARP	`struct arpreq *`	Delete an ARP protocol address (IP address/hardware address pair) entry from the translation table.

Examples

See UDP Client Program (page 219) for examples that call the `socket_ioctl` function.

The following program excerpt shows an example of using both the `SIOCGIFCONF` and `SIOCGIFNUM` functions. The names of all interfaces configured are displayed.

```
... /* declarations */
struct ifreq* ifr;
struct ifconf* ifc;
int            ifcount,res,datasize,bufsize,i,ifr_count;
... /* procedure code */
... /* assume socket is already created, descriptor 'sd' */
res = socket_ioctl(sd,SIOCGIFNUM, (char*)&ifcount);
... /* error checking */
/* bufsize * 2 to allow for alias entries */
datasize = sizeof(struct ifreq) * ifcount * 2;
bufsize = sizeof(struct ifconf) + datasize;
ifc = (struct ifconf*)malloc(bufsize);
... /* error checking */
ifc->ifc_len = datasize;
ifc->ifc_buf = (char*)&ifc[1];
res = socket_ioctl(sd,SIOCGIFCONF,(char*)ifc);
... /* error checking */
ifr_count = ifc->ifc_len / sizeof(struct ifreq);
ifr = (struct ifreq*)&ifc[1];
for (i=0; i<ifr_count;i++)
   printf("Interface %d: %s\n",i,ifr[i].ifr_name);
.../*end of program extract*/
```

socket_set_inet_name

The `socket_set_inet_name` function specifies the name of the NonStop TCP/IP or TCP6SAM process that the socket library is going to open.

C Synopsis

```
#include "netdb.h"

void socket_set_inet_name (name_ptr);

       char *name_ptr;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

void socket_set_inet_name (name_ptr);

       STRING .EXT name_ptr;
```

name_ptr
input value; points to a null-terminated character string containing the process name of the NonStop TCP/IP or TCP6SAM process that is to be accessed by subsequent calls to `socket` or `socket_nw`.

Errors

No errors are returned for this function.

Usage Guidelines

The `socket` or `socket_nw` function opens the NonStop TCP/IP, TCP6SAM or CIPSAM process by name. Therefore, the function must know the name of this process. If your program calls the `socket_set_inet_name` function before calling the `socket` or `socket_nw` function, the socket library opens the TCP/IP process you specified.

If your program does not call `socket_set_inet_name`, the socket library opens the process with the name defined for `=TCPIP^PROCESS^NAME DEFINE`. If a defined name does not exist, the socket library uses the default process `$ZTC0`. For more information on `=TCPIP^PROCESS^NAME`, see Using the DEFINE Command (page 29).

NOTE: Name resolver socket API calls (for example, `gethostbyname`, `gethostbyaddr`, `getaddrinfo`, and so on) access the TCP/IP stack through the TCP/IP socket library (which makes the initial `socket` or `socket_nw` call). The TCP/IP stack that is used by these socket library calls is assigned by either the `=TCPIP^PROCESS^NAME TACL DEFINE` or by the `socket_set_inet_name` socket API call.

t_recvfrom_nw

The `t_recvfrom_nw` function receives data on an unconnected UDP socket or raw socket created for nowait operations. This routine is replaced by the `recvfrom_nw` routine.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <netdb.h>

error = t_recvfrom_nw (socket, r_buffer_ptr, length,
                       flags, tag );
        int socket, length, error, flags;
        struct sendto_recvfrom_buf *r_buffer_ptr;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := recvfrom_nw (socket, r_buffer_ptr, length,
                      flags, tag );

        INT          socket,
                        length,
                        flags;
        INT .EXT     r_buffer_ptr(sendto_recvfrom_buf);
        INT(32)      tag;
```

error
: return value; if the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 202).

socket
: input value; specifies the socket number for the socket, as returned by the call to `socket_nw`.

r_buffer_ptr
: input and return value; on completion, points to the remote address and port number from which the data is received, followed by the data. The address of the data is `(r_buffer_ptr + sizeof(struct sockaddr_in))`, where `sizeof(struct sockaddr_in)` is 16 bytes.

length
: input value; the size of the buffer pointed to by *r_buffer_ptr*. The size of the buffer is the size of the data plus `sizeof(struct sockaddr_in)`, where `sizeof(struct sockaddr_in)` is 16 bytes.

flags
: input value; specifies how the incoming message is to be read, and is one of the following values:

MSG_PEEK	Read the incoming message without removing the message from the queue.
0	No flag; read data normally.

tag
: is the *tag* parameter to be used for the nowait operation initiated by `t_recvfrom_nw`.

Errors

If an error occurs, the return value is set to -1 and the external variable errno is set to one of the following values:

`EISCONN`	The specified socket was connected.
`ESHUTDOWN`	The specified socket was shut down.
`EINVAL`	An invalid argument was specified.

Usage Guidelines

- This is a nowait call; it must be completed with a call to the `AWAITIOX` procedure. For a waited call, use `recvfrom`.
- The parameters of the `t_recvfrom_nw` function are not compatible with those of the `recvfrom` function in the 4.3 BSD UNIX operating system.

- The length of the received data is given in the third parameter (count transferred) returned from the AWAITIOX procedure. This length includes the address information given by sizeof(sockaddr_in) at the beginning of the buffer.
- Note that the MSG_OOB option is not available. This is a constraint imposed by UDP. UDP does not support out-of-band data.

See Nowait Call Errors (page 86) for information on checking errors.

t_recvfrom_nw64_

The t_recvfrom_nw64_ function receives data on an unconnected UDP socket or raw socket created for nowait operations. This routine is replaced by the recvfrom_nw64_ routine.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <netdb.h>

error = t_recvfrom_nw64_ (socket, r_buffer_ptr64, length,
                          flags, tag );
        int socket, length, error, flags;
        struct sendto_recvfrom_buf _ptr64 *r_buffer_ptr64;
        long long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := t_recvfrom_nw64_ (socket, r_buffer_ptr64, length,
                           flags, tag );

        INT          socket,
                        length,
                        flags;
        INT .EXT64     r_buffer_ptr64(sendto_recvfrom_buf);
        INT(64)      tag;
```

error
: return value; if the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call fails, the external variable *errno* is set as shown in Errors (page 204).

socket
: input value; specifies the socket number for the socket, as returned by the call to socket_nw.

r_buffer_ptr64
: input and return value; on completion, points to the remote address and port number from which the data is received, followed by the data. The address of the data is (r_buffer_ptr64 + sizeof(struct sockaddr_in)), where sizeof(struct sockaddr_in) is 16 bytes.

length
: input value; the size of the buffer pointed to by *r_buffer_ptr64*. The size of the buffer is the size of the data plus sizeof(struct sockaddr_in), where sizeof(struct sockaddr_in) is 16 bytes.

flags
input value; specifies how the incoming message must be read, and takes one of the following values:

MSG_PEEK	Read the incoming message without removing the message from the queue.
0	No flag; read data normally.

tag
is the *tag* parameter to be used for the nowait operation initiated by `t_recvfrom_nw64_`.

Errors

If an error occurs, the return value is set to -1 and the external variable errno is set to one of the following values:

`EISCONN`	The specified socket was connected.
`ESHUTDOWN`	The specified socket was shut down.
`EINVAL`	An invalid argument was specified.

Usage Guidelines

- This is a nowait call; it must be completed with a call to the `FILE_AWAITIO64_` procedure. For a waited call, use `recvfrom64_`.
- The parameters of the `t_recvfrom_nw64_` function are not compatible with those of the `recvfrom64_` function in the 4.3 BSD UNIX operating system.
- The length of the received data is specified in the third parameter (count transferred) returned from the `FILE_AWAITIO64_` procedure. This length includes the address information given by `sizeof(sockaddr_in)` at the beginning of the buffer.
- Note that the MSG_OOB option is not available. This is a constraint imposed by UDP. UDP does not support out-of-band data.

For information on checking errors, see Nowait Call Errors (page 86).

t_sendto_nw

The `t_sendto_nw` function sends data on an unconnected UDP socket or raw socket created for nowait operations. This routine is replaced by the `sendto_nw` routine.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <netdb.h>

error = t_sendto_nw (socket, r_buffer_ptr, length, flags, tag);

        int error, socket, length, flags;
        struct sendto_recvfrom_buf *r_buffer_ptr;
        long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := t_sendto_nw (socket, r_buffer_ptr, length, flags, tag);

        INT socket,
            length;
```

```
        flags
INT .EXT sockaddr_ptr(sockaddr);
INT(32) tag;
```

error
: return value; if the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call failed, the external variable *errno* is set as indicated in Errors (page 205).

socket
: input value; specifies the socket number for the socket, as returned by a `socket_nw` call.

r_buffer_ptr
: input and return value; points to the remote address and port number to which the data is to be sent, followed by the data. The address of the data is `(r_buffer_ptr + sizeof(struct sockaddr_in))`. See the `sendto_recvfrom_buf` structure in "Data Structures".

  Note that the first two bytes pointed to by *r_buffer_ptr* are the `sin_family` field of the `sockaddr_in` structure. After a call to `t_sendto_nw`, the normal value in the `sin_family` field `(AF_INET)` is replaced by the number of bytes that have been transferred.

length
: input value; the size of the buffer pointed to by *r_buffer_ptr*.

flags
: input value; specifies whether the outgoing data should be sent to the destination if routing is required. This parameter can be one of the following values:

`MSG_DONTROUTE`	Send this message only if the destination is located on the local network; do not send the message through a gateway.
`0`	No flag; send the message to the destination, even if the message must be routed.

tag
: input value; the *tag* parameter to be used for the nowait operation initiated by `t_sendto_nw`.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

`EMSGSIZE`	The message was too large to be sent atomically, as required by the socket options.
`EISCONN`	The specified socket was connected.
`ESHUTDOWN`	The specified socket was shut down.
`ENETUNREACH`	The destination network was unreachable.
`EINVAL`	An invalid argument was specified.

Usage Guidelines

- This is a nowait call; it must be completed with a call to the `AWAITIOX` procedure. For a waited call, use `sendto`.
- The parameters of the `t_sendto_nw` function are not compatible with those of the `sendto` function in the 4.3 BSD UNIX operating system.
- To determine the number of bytes transferred as a result of the `t_sendto_nw` function, check the `sb_sent` field of the `sendto_recvfrm_buf` structure. This field is defined the same as the `sin_family` field of the `sockaddr_in` structure. After you use this value, reset the `sin_family` field to `AF_INET`.

See Nowait Call Errors (page 86) for information on error checking.

t_sendto_nw64_

The t_sendto_nw64_ function sends data on an unconnected UDP socket or raw socket created for nowait operations. This routine is replaced by the sendto_nw64_ routine.

C Synopsis

```
#include <socket.h>
#include <in.h>
#include <netdb.h>

error = t_sendto_nw64_ (socket, r_buffer_ptr64, length, flags, tag);

        int error, socket, length, flags;
        struct sendto_recvfrom_buf _ptr64 *r_buffer_ptr64;
        long long tag;
```

TAL Synopsis

```
?NOLIST, SOURCE SOCKDEFT
?NOLIST, SOURCE SOCKPROC

error := t_sendto_nw64_ (socket, r_buffer_ptr64, length, flags, tag);

        INT socket,
              length,
              flags;
        INT .EXT64     r_buffer_ptr64(sendto_recvfrom_buf);
        INT .EXT64 sockaddr_ptr64(sockaddr);
        INT(32) tag;
```

error
return value; if the call is successful, a zero is returned. If the call is not successful, –1 is returned. If the call fails, the external variable *errno* is set as shown in Errors (page 207).

socket
input value; specifies the socket number for the socket, as returned by a socket_nw call.

r_buffer_ptr64
input and return value; points to the remote address and port number to which the data must be sent, followed by the data. The address of the data is (*r_buffer_ptr64* + sizeof(struct sockaddr_in)). For more information, see the sendto_recvfrom_buf structure in Data Structures (page 63).

Note that the first two bytes pointed to by *r_buffer_ptr64* are the sin_family field of the sockaddr_in structure. After a call to t_sendto_nw64_, the normal value in the sin_family field (AF_INET) is replaced by the number of bytes that have been transferred.

length
input value; the size of the buffer pointed to by *r_buffer_ptr64*.

flags
input value; specifies whether the outgoing data must be sent to the destination if routing is required, and takes one of the following values:

MSG_DONTROUTE	Send this message only if the destination is located on the local network; do not send the message through a gateway.
0	No flag; send the message to the destination, even if the message must be routed.

tag
input value; the *tag* parameter to be used for the nowait operation initiated by t_sendto_nw64_.

Errors

If an error occurs, the external variable *errno* is set to one of the following values:

EMSGSIZE	The message was too large to be sent atomically, as required by the socket options.
EISCONN	The specified socket was connected.
ESHUTDOWN	The specified socket was shut down.
ENETUNREACH	The destination network was unreachable.
EINVAL	An invalid argument was specified.

Usage Guidelines

- This is a nowait call; it must be completed with a call to the FILE_AWAITIO64_ procedure. For a waited call, use sendto64_.
- The parameters of the t_sendto_nw64_ function are not compatible with those of the sendto64_ function in the 4.3 BSD UNIX operating system.
- To determine the number of bytes transferred as a result of the t_sendto_nw64_ function, check the sb_sent field of the sendto_recvfrm_buf structure. This field has the same definition as the sin_family field of the sockaddr_in structure. After you use this value, reset the sin_family field to AF_INET.

For information on error checking, see Nowait Call Errors (page 86).

5 Sample Programs

This section provides TCP/IP program examples for AF_INET sockets and AF_INET6 sockets.

Programs Using AF_INET Sockets

This subsection contains a client and server program that use AF_INET sockets.

AF_INET Client Stub Routine

The first example shows a sample client program that you can build, compile, and run on your system. The program sends a request to and receives a response from the system specified on the command line.

```
/*
 *      AF_INET Client Stub Routine
 * ****************************************************************
 * *                                                              *
 * *    Copyright (c) Hewlett-Packard Company, 2003               *
 * *                                                              *
 * *   The software contained on this media  is  proprietary  to  *
 * *   and  embodies  the  confidential  technology  of  Hewlett  *
 * *   Packard Corporation.  Possession, use,  duplication  or    *
 * *   dissemination of the software and media is authorized only *
 * *   pursuant to a valid written license from Hewlett Packard   *
 * *   Corporation.                                               *
 * *                                                              *
 * *   RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure *
 * *   by the U.S. Government is subject to restrictions  as  set *
 * *   forth in Subparagraph (c)(1)(ii)  of  DFARS  252.227-7013, *
 * *   or  in  FAR 52.227-19, as applicable.                      *
 * *                                                              *
 * ****************************************************************
 */
#include <systype.h>
#include <socket.h>
#include <errno.h>
#include <in.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <inet.h>
#include <cextdecs(FILE_CLOSE_)>

#define SERVER_PORT 7639
#define CLIENT_PORT 7739

#define MAXBUFSIZE 4096

int main (int argc, char **argv )
{
    int             s;
    int             error;
    char            databuf[MAXBUFSIZE];
    int             dcount;
    const char      *ap;
    struct hostent  *hp;
    char            *server;

    /* Declare sockaddr_in structures for IPv4 use.*/
    struct sockaddr_in     serveraddr;
```

```
  char            request[MAXBUFSIZE] = " This is the client's request";
  if (argc < 2) {
      printf("Usage: client <server>\n");
      exit (0);
  }
  server = argv[1];

  /* Clear the server address and sets up server variables.
     The socket address is a 32-bit Internet address and a 16-bit
     port number. */
  bzero((char *) &serveraddr, sizeof(struct sockaddr_in));
  serveraddr.sin_family = AF_INET;

/* Obtain the server's IPv4 address. A call to gethostbyname
   returns IPv4 address only. */
  if ((hp = gethostbyname(server)) == NULL) {
      printf("unknown host: %s\n", server);
    exit(2);
  }
  serveraddr.sin_port = htons(SERVER_PORT);

 /* Creates an AF_INET socket with a socket call. The socket type
    SOCK_STREAM is specified for TCP or connection-oriented
    communication. */
  while (hp->h_addr_list[0] != NULL) {
    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket");
      exit(3);
    }
    memcpy(&serveraddr.sin_addr.s_addr, hp->h_addr_list[0],
                       hp->h_length);

    /* Connect to the server using the address in the sockaddr_in
       structure named serveraddr. */
    if ((error = connect(s, (struct sockaddr *)&serveraddr,
                         sizeof(serveraddr)) ) < 0) {
      perror("connect");
      hp->h_addr_list++;
      continue;
    }
    break;
  }
  if (error < 0)
    exit(4);

  /* Send a request to the server. */
  if (send(s, request, (int)strlen(request), 0) < 0) {
    perror("send");
    exit(5);
  }

  /* Receive a response from the server. */
  dcount = recv(s, databuf, sizeof(databuf), 0);
  if (dcount < 0) {

      perror("recv");
      exit(6);
  }
  databuf[dcount] = '\0';

   /* Get the server name using the address in the sockaddr_in
      structure named serveraddr. A call to gethostbyaddrexpects an
      IPv4 address as input. */
  hp = gethostbyaddr((char *)&serveraddr.sin_addr.s_addr,
      sizeof(serveraddr.sin_addr.s_addr), AF_INET);
```

```
    /* Convert the server's 32-bit IPv4 address to a dot-formatted
       Internet address text string. A call to inet_ntoa expects an
       IPv4 address as input. */
    ap = inet_ntoa(serveraddr.sin_addr);
    printf("Response received from");
    if (hp != NULL)
      printf(" %s", hp->h_name);
    if (ap != NULL)
      printf(" (%s)", ap);
    printf(":\n %s\n", databuf);
    FILE_CLOSE_((short)s);
}
```

AF_INET Server Stub Routine

The next example shows a sample server program that you can build, compile, and run on your system. The program receives requests from and sends responses to client programs on other systems.

```
/*
 *      AF_INET Server Stub Routine
 * ****************************************************************
 * *                                                              *
 * *    Copyright (c) Hewlett-Packard Company, 2003               *
 * *                                                              *
 * *   The software contained on this media  is  proprietary  to  *
 * *   and  embodies  the  confidential  technology  of  Hewlett  *
 * *   Packard  Corporation.  Possession, use,  duplication  or   *
 * *   dissemination of the software and media is authorized only *
 * *   pursuant to a valid written license from Hewlett Packard   *
 * *   Corporation.                                               *
 * *                                                              *
 * *   RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure *
 * *   by the U.S. Government is subject to restrictions  as  set *
 * *   forth in Subparagraph (c)(1)(ii)  of  DFARS  252.227-7013, *
 * *   or  in  FAR 52.227-19, as applicable.                      *
 * *                                                              *
 * ****************************************************************
 */

#include <systype.h>
#include <socket.h>
#include <errno.h>
#include <in.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <inet.h>
#include <cextdecs(FILE_CLOSE_)>

#define SERVER_PORT     7639
#define CLIENT_PORT     7739

#define MAXBUFSIZE 4096

int main (
    int argc,
    char **argv )

{
    int             s;
    char            databuf[MAXBUFSIZE];
```

```
int             new_s;
int             dcount;
u_short         port;
struct hostent  *hp;
const char      *ap;

/* Declares sockaddr_in structures. The use of this type of
 structure implies communication using the IPv4 protocol. */
struct sockaddr_in    serveraddr;
struct sockaddr_in    clientaddr;
int                   clientaddrlen;
char            response[MAXBUFSIZE] = " This is the server's response";

/* Creates an AF_INET socket. The socket type SOCK_STREAM is
   specified for TCP or connection-oriented communication. */
if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("socket");
  exit (0);
}

/* Clear the server address and sets up server variables. The socket
   address is a 32-bit Internet address and a 16-bit port number on
   which it is listening.*/
bzero((char *) &serveraddr, sizeof(struct sockaddr_in));
serveraddr.sin_family      = AF_INET;

/* Set the server address to the IPv4 wild card address
   INADDR_ANY. This signifies any attached network interface on
   the system. */
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port        = htons(SERVER_PORT);
if (bind(s, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) < 0) {
/* Binds the server's address to the AF_INET socket. */
  perror("bind");
  exit(2);
}
while (1) {

  clientaddrlen = sizeof(clientaddr);

 /*Accept a connection on this socket. The accept call places the
   client's address in the sockaddr_in structure named clientaddr. */
  new_s = accept(s, (struct sockaddr *)&clientaddr, &clientaddrlen);
    if (new_s < 0) {
    perror("accept");
    continue;
  }
  /* Receive data from the client. */
  dcount = recv(new_s, databuf, sizeof(databuf), 0);
  if (dcount <= 0) {
    perror("recv");
    FILE_CLOSE_((short)new_s);
    continue;
  }
  databuf[dcount] = '\0';

  /* Retrieve the client name using the address in the sockaddr_in
     structure named clientaddr. A call to gethostbyaddr expects an
     IPv4 address as input. */

  hp = gethostbyaddr((char *)&clientaddr.sin_addr.s_addr,
          sizeof(clientaddr.sin_addr.s_addr), AF_INET);

   /* Convert the client's 32-bit IPv4 address to a dot-formatted
      Internet address text string. A call to inet_ntoa expects an
```

```
        IPv4 address as input. */
      ap = inet_ntoa(clientaddr.sin_addr);
      port = ntohs(clientaddr.sin_port);
      printf("Request received from");
      if (hp != NULL)
        printf(" %s", hp->h_name);
      if (ap != NULL)
        printf(" (%s)", ap);
      printf(" port %d\n\"%s\"\n", port, databuf);

       /* Send a response to the client. */

      if (send(new_s, response, (int)strlen(response), 0) < 0)
{
        perror("send");
        FILE_CLOSE_((short)new_s);
        continue;
      }
      FILE_CLOSE_((short)new_s);
    }
    FILE_CLOSE_((short)s);
}
```

AF_INET No-Wait Server Stub Routine

```
/*
* AF_INET Server Stub Routine
* ****************************************************************
* *                                                              *
* *   Copyright (c) Hewlett-Packard Company, 2003                *
* *                                                              *
* *   The software contained on this media is proprietary to     *
* *   and embodies the confidential technology of Hewlett        *
* *   Packard Corporation. Possession, use, duplication or       *
* *   dissemination of the software and media is authorized only *
* *   pursuant to a valid written license from Hewlett Packard   *
* *   Corporation.                                               *
* *                                                              *
* *   RESTRICTED RIGHTS LEGEND Use, duplication, or disclosure   *
* *   by the U.S. Government is subject to restrictions as set   *
* *   forth in Subparagraph (c)(1)(ii) of DFARS 252.227-7013,    *
* *   or in FAR 52.227-19, as applicable.                        *
* *                                                              *
* ****************************************************************
*/

/*  This is the same as the IPV4 sample server, but using nowaited I/O calls */

#include <systype.h>
#include <socket.h>
#include <errno.h>
#include <in.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <inet.h>
#include <tal.h>
#include <ctype.h>
#include <cextdecs.h>
#define SERVER_PORT 7639
#define CLIENT_PORT 7739
#define MAXBUFSIZE 4096

long    tagBack;

short   completedSocket;
short   dcount;
```

```
short IOCheck ( long TOVal ) {
    /* use a single AWAITIOX() check for all I/O in this pgm
       return value is FE;
       sets global tagBack & socket that completed;
       don't care about buf addr but do want count */
    short   error;
    _cc_status  CC;

    completedSocket = -1;
    CC = AWAITIOX( &completedSocket,,&dcount,&tagBack,TOVal );
    /* ignoring possible _status_gt condition */
    if( _status_lt( CC ) ) {
        FILE_GETINFO_( completedSocket,&error );
        return error;
    }
    else return 0;
}

int main (int argc,char **argv ) {
    int     s;
    char    databuf[MAXBUFSIZE];
    int     new_s;
    u_short port;
    struct  hostent *hp;
    const char *ap;
    short   fe;
    long    tag = 44;       /* for nowait I/O ID */
    long    tag2 = 45;      /*   "      "        */
    long    acceptWait = -1;/* how long to wait for connections */
    long    timeout = 500;  /* read t/o of 5 secs */

    /* Declares sockaddr_in structures. The use of this type of
       structure implies communication using the IPv4 protocol. */
    struct sockaddr_in serveraddr;
    struct sockaddr_in clientaddr;
    int clientaddrlen;
    char response[MAXBUFSIZE] = " This is the server's response";

    /* Create an AF_INET socket.
        FLAGS argument does not indicate open nowait (octal 200) ,
        but does indicate 2 outstanding I/Os max.
        SETMODE 30 included in the call */
    if ((s = socket_nw(AF_INET, SOCK_STREAM, 0, 2, 0)) < 0) {
            perror("socket");
            exit (0);
    }

    /* Clear the server address and set up server variables. The socket
        address is a 32-bit Internet address and a 16-bit port number on
        which it is listening.*/
    bzero((char *) &serveraddr, sizeof(struct sockaddr_in));
    serveraddr.sin_family = AF_INET;

    /* Set the server address to the IPv4 wild card address
        INADDR_ANY. This signifies any attached network interface on
        the system. */
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons(SERVER_PORT);

    /* Bind the server's address to the AF_INET socket. */
    if (bind_nw(s, (struct sockaddr *)&serveraddr, sizeof(serveraddr), tag)<0){

            perror("bind");
            exit(2);
    }

    if( fe = IOCheck( -1 ) ) {
            printf( "AWAITIO error %d from bind_nw\n",fe );
            exit(2);
    }
```

```
while (1) {
    /*  Accept a connection on this socket. The accept call places the
        client's address in the sockaddr_in structure named clientaddr.*/
    clientaddrlen = sizeof(clientaddr);
    if( accept_nw(s, (struct sockaddr *)&clientaddr, &clientaddrlen, tag) <0) {
            perror("accept");
            exit(3);
    }

    if( fe = IOCheck(acceptWait)  ) {       /* initially, wait -1;
                                               maybe change afterwards? */
        if( fe == 40 ) {
            printf( "Timed out after %ld secs wtg Client connect.
                                          Terminating.\n",acceptWait/100 );
            FILE_CLOSE_((short)s);
            exit(0);
        } else {
            printf( "AWAITIO error %d from accept_nw\n",fe );
            exit(3);
        }
    }

    /* Need a new socket for the data transfer
        Resembles the earlier call */
    if ((new_s = socket_nw(AF_INET, SOCK_STREAM,0,2,0)) < 0) {
        perror ("Socket 2 create failed.");
        exit (4);
    }

    /* Make the connection */
    if ( accept_nw2(new_s, (struct sockaddr *)&clientaddr, tag2) < 0) {
        perror ("2nd Accept failed.");
        exit (5);
    }

    if( fe = IOCheck(-1)  ) {
        printf( "AWAITIO error %d, tag %ld from 2nd
                                            accept_nw\n",fe,tagBack );
        exit(4);
    }

    /* Receive data from the client.
       recv_nw() - awaitio() should be in a loop until a logical record
       has been received.  In this example, we expect the short messages
       to be completed in a single recv_nw() */
    if( recv_nw(new_s, databuf, sizeof(databuf), 0, tag2) < 0 ) {
        if( errno == ESHUTDOWN || errno == ETIMEDOUT || errno ==
                                                        ECONNRESET ) {
            FILE_CLOSE_((short)new_s);
            continue;
        } else {
            perror( "recv_nw error" );
            exit( 6 );
        }
    }

    if( fe = IOCheck(timeout) ) {
        if( fe == 40 ) {         /* abandon and start over */
            FILE_CLOSE_((short)new_s);
            continue;
        } else {
            printf( "AWAITIO error %d from recv_nw\n",fe );
            exit(6);
        }
    }

    databuf[dcount] = '\0';     /* dcount set by IOCheck */

    /* Retrieve the client name using the address in the sockaddr_in
        structure named clientaddr. A call to gethostbyaddr expects an
        IPv4 address as input. */
```

```
        hp = gethostbyaddr((char *)&clientaddr.sin_addr.s_addr,
        sizeof(clientaddr.sin_addr.s_addr), AF_INET);

        /* Convert the client's 32-bit IPv4 address to a dot-formatted
            Internet address text string. A call to inet_ntoa expects an
            IPv4 address as input. */
        ap = inet_ntoa(clientaddr.sin_addr);
        port = ntohs(clientaddr.sin_port);

        printf("Request received from");
        if (hp != NULL) printf(" %s", hp->h_name);
        if (ap != NULL) printf(" (%s)", ap);
        printf(" port %d\n\"%s\"\n", port, databuf);

        /* Send a response to the client. */
        if (send_nw2(new_s, response, (int)strlen(response), 0, tag2) < 0) {
            perror("send_nw2");
            FILE_CLOSE_((short)new_s);
            continue;
        }

        if( fe = IOCheck( -1 ) ) {
            FILE_CLOSE_((short)new_s);
            continue;
        }
    }    /* while */
}/*
```

C TCP Client Program

The following client program on one NonStop system sends data from its memory to the server on another NonStop system, where the two hosts are connected over a network or an internetwork:

To compile the program in native mode, run this command:

```
> nmc/in <input file name>,out <list file name>/<object file
name>;symbols,runnable,extensions,ssv0 "subvolume name",ssv1
"$system.system",ssv2
"$system.zsysdefs",ssv3 "$system.ztcpip"
```

NOTE: Before running the client program, create a send file with object code 000.

To run the client program:

```
> run <objectfile name> <send file name> <host port #> <process name>
```

Sample Program

```
#pragma nolist
#include <cextdecs(FILE_CLOSE_,read)>
#include <unistd.h>
#include <param.h>
#include <socket.h>
#include <in.h>
#include <netdb.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <memory.h>
#include <errno.h>
#define INET_ERROR  4294967295 /* inet_addr returns 0xffffffffl upon error */
#pragma list

/*
 * Usage: CLIENT send_file host port# proc_name nbufs bufsize
 */

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

```
/* define things */

        int fo;
        int rdstat,nbytes;

        register int fd;
        struct sockaddr_in sin;
        char *buf;
        char *procname;
        int nbufs, bsize;
        int port;
        struct hostent *host_entry;

        /* DEBUG(); */

/* open send file */

        argc--; argv++;
        if (argc < 3)
                goto usage;
        if ((fo = (open(argv[0],O_RDONLY))) < 0) {
                printf ("CLIENT: open failed\n");
                exit(0);
        }

/* set address according to device name */

        argc--; argv++;
        if ((sin.sin_addr.s_addr = inet_addr(argv[0])) == INET_ERROR ){
           if ((host_entry = gethostbyname(argv[0])) ==
                        (struct hostent *)NULL) {
                printf ("Get host by name failed, error %d\n",h_errno);
                exit(0);
            }
        sin.sin_addr.s_addr =
              *(unsigned long *) (*(host_entry->h_addr_list));
        }
        else
           sin.sin_addr.s_addr = inet_addr(argv[0]);

/* set port number */

        argc--; argv++;
        if ((port = atoi (argv [0])) <= 0)
                goto usage;

/* set the process name */

        argc--; argv++;
        if (argc > 0)
                procname = argv[0];
        else
                procname = "$ZTC0";

/* set the number of buffers to be sent */

        argc--; argv++;
        if (argc > 0)
                nbufs = atoi (argv [0]);
        else
                nbufs = 1;

/* set the size of the buffer to be sent */

        argc--; argv++;
        if (argc > 0)
                bsize = atoi (argv [0]);
        else
                bsize = 1024;

        buf = (char *)malloc (bsize);
```

```
        nbytes = bsize;

/* lets open the process */


        printf ("CLIENT: Data is sent with TCPIP process %s \n",procname);
        (void) socket_set_inet_name (procname);

/* lets open the socket */

        if ((fd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
               perror ("CLIENT: socket");

                exit(0);
        }
        printf ("CLIENT: Socket # %d opened ... \n", fd);

        sin.sin_family = AF_INET;
        sin.sin_port = (unsigned short)port;
        if (connect (fd,(struct sockaddr *)&sin, (int)(sizeof (sin))) < 0) {
            /*  printf ("CLIENT: errno is %s \n",errno);  */
               perror ("CLIENT: connect");
                exit(0);
        }
        printf ("CLIENT: Connected ...\n");

        while (nbufs-- > 0) {
                int sent, tosend;
                sent = 0;
                rdstat = (read(fo,buf,nbytes));
                printf ("CLIENT: Bytes read from file %d \n",rdstat);
                tosend = rdstat;
                if (rdstat > 0) {
                   retry:
                   if ((sent=send (fd, (buf + sent), tosend, 0)) < 0) {
                       perror ("CLIENT: send");
                        exit(0);
                   }
                   printf ("CLIENT: sent %d bytes\n",sent);
                   if (sent < tosend) {
                        tosend -= sent;
                        printf ("CLIENT: sending more data ...\n");
                        goto retry;
                   }
                 } else nbufs=0;
        }   /* while */
        printf ("CLIENT: Send completed.\n");
        FILE_CLOSE_((short int)fo);
        exit(0);
        usage:
          fprintf (stderr, "usage:CLIENT send_file host port# proc_name");
          fprintf (stderr, " nbufs bufsize \n");
exit(0);
}
```

C TCP Server Program

The following server program receives data from the previous client program. To run this server with default port 25, you must be logged on as a SUPER user.

To compile the program in native mode, run this command:

```
> nmc/in <input file name>,out <list file name>/<object file
name>;symbols,runnable,extensions,ssv0 "subvolume name",ssv1
"$system.system",ssv2
"$system.zsysdefs",ssv3 "$system.ztcpip"
```

NOTE: Before running the server program, create a receive file with object code 101.

To run the server program:

```
> run <objectfile name ><receive file name><port #><process name >
```

Sample Program

```
#pragma nolist
#include <$system.ztcpip.param.h>
#include <$system.ztcpip.socket.h>
#include <$system.ztcpip.in.h>
#include <$system.ztcpip.netdb.h>
#include <stdio.h>

#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#include <stdlib.h>
#include <errno.h>
#include <cextdecs(TIME,CLOSE,FILE_CLOSE_,WRITE)>
#pragma list

/*
 * Usage: SERVER recv_file port# proc_name
 */
long state, total_read;
char buf [12000/(int)(sizeof (char) + 1)];
int sizebuf = (12000/(int)(sizeof (char) + 1));

main (argc, argv)
        int argc;
        char *argv[];
{
        int fo, wc;
        int nnnn = 2340;
        register int fd, s2, cc;
        int flen = 8, port;
        struct sockaddr_in sin, from;
        char *procname;

/* open receive file */

        argc--; argv++;
        if (argc < 2)
                goto usage;
        if ((fo = (open(argv[0],O_RDWR|O_CREAT|O_TRUNC,nnnn))) < 0) {
                printf ("SERVER: open failed\n");
                exit(0);
        }

/* Set the port address */

        argc--; argv++;
        if ((port = atoi (argv[0])) <= 0)
                goto usage;

/* set the process name */

        argc--; argv++;
        if (argc > 0)
                procname = argv[0];
        else
                procname = "$ZTC0";

/* lets open the process */

        printf ("SERVER: Data is recd with Tandem NonStop TCP/IP process %s\n",procname);
        (void) socket_set_inet_name (procname);

/* Open the socket */

        if ((fd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
                fprintf (stderr, "SERVER: socket-failure (%d)\n", errno);
                exit (0);
        }
        printf ("SERVER: Socket # %d opened ...\n", fd);

/* Set up sin.x values */

        sin.sin_family = AF_INET;
        sin.sin_addr.s_addr = INADDR_ANY;
        sin.sin_port = (unsigned short)port;
```

```
/* Bind the socket */

        if (bind (fd,(struct sockaddr *)&sin, (int)sizeof (sin)) < 0) {
                perror ("SERVER: bind");
                exit (0);

        }

        printf ("SERVER: BIND completed ...\n");

        if (listen (fd, 5) < 0) {
                perror ("SERVER: listen");
                exit (0);
        }
        printf ("SERVER: Listening on socket # %d \n", fd);

        if ((s2 = accept (fd,(struct sockaddr *)&from, &flen)) < 0) {
                perror ("SERVER: accept");
                exit (0);
        }
        printf ("SERVER: Connected ...\n");
        total_read = 0;
        while ((cc = recv (s2, buf, sizebuf,0)) > 0) {
                printf ("SERVER: read %d bytes ... \n",cc);
                total_read += (long)cc;
                printf ("SERVER: copying buffer to file ...  \n");
                if ((wc=write(fo,buf,cc)) <0) {
                        printf ("SERVER: write failed\n");
                        exit(0);
                        }
                else
                        printf ("SERVER: copied %d bytes \n",wc);
        }
        (void) FILE_CLOSE_ ((short int)s2);
        printf ("SERVER: Receive completed.\n");
        FILE_CLOSE_((short int)fo);
        exit(0);
        usage:
          fprintf(stderr, "usage: SERVER recv_file port proc_name\n");
        exit(0);
}
```

Client and Server Programs Using UDP

This subsection contains a client and a server program that demonstrate a UDP communication. The client on one NonStop system sends a string of characters entered by a user to the server on another NonStop system. The server sends (echoes) the string back to the client.

TIP: When using the NonStop TCP/IPv6 network mode to call the `socket_ioctl` function, you must configure the “Family” attribute to “DUAL” in the PROVIDER object (associated with the CIPSAM process). If the Family attribute is set to "INET", all NonStop TCP/IPv6 addresses are ignored and not returned to the socket_ioctl caller. When the attribute is set to DUAL, the NonStop TCP/IPv6 addresses are returned, but the size of the entries are variable and based on the actual address type:

- For a NonStop TCP/IP address, `IFNAMSIZ=sizeof(struct sockaddr)` bytes is passed back.
- For a NonStop TCP/IPv6 address, `IFNAMSIZ=sizeof(struct sockaddr_in6)` bytes is passed back.

UDP Client Program

The following programming example shows how to use the socket routines in a UDP client application using the NonStop TCP/IP network mode:

```
#pragma nolist
#include <$system.ztcpip.param.h>
#include <$system.ztcpip.socket.h>
#include <$system.ztcpip.ioctl.h>
#include <$system.ztcpip.in.h>
#include <$system.ztcpip.netdb.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>
#include <cextdecs(DELAY)>
#define INET_ERROR 4294967295
#pragma list

/*
 * The following DEFINES control the behavior of the client.
 */

#define CONNECTIONLESS /* Do not connect to host that sends you packet */
#define DONTROUTE       /* Tell IP not to use routing to send this packet */
#define BROADCAST       /* Tell IP to allow broadcasting of this packet */
#define SETBUF          /* Set Receive and Send buffer sizes */
#define PORT_ECHO 1987
int channel;

main (argc, argv)
int     argc;
char    *argv[];
{
     struct   sockaddr_in remote, him, me;
     int status, len, ncc, tosend;
     int optval, optlen;
     long haddr;
     char buffer[8*1024];
     struct hostent *hp;
     if (argc < 2)  {
          printf ("Usage: %s hostname\n", *argv);
          exit (0);
     }

     /*
      * Get the host address of the remote server
      */

     if ( (haddr = (long)inet_addr(argv[1])) == INET_ERROR ) {
        if ((hp = gethostbyname(argv[1])) == (struct hostent *)NULL) {
          printf ("%s: unknown host\n", argv[1]);
          exit (0);
        }
        bcopy (hp->h_addr, (char *)&remote.sin_addr.s_addr, hp->h_length);
     }
     else
        remote.sin_addr.s_addr = haddr;
         remote.sin_family = AF_INET;
         remote.sin_port = htons(PORT_ECHO);

     /*
      * Create a socket
      */

     channel = socket(AF_INET, SOCK_DGRAM, 0);
     if (channel == -1) {
          printf ("echo client: socket failed\n");
          exit (0);
     }
printf("Socket -client created\n");

#ifdef BROADCAST
     printf("\nExecute SETSOCKOPT to allow broadcasting\n");
     optlen = sizeof(optval);
     optval = 1;
     if (setsockopt(channel,SOL_SOCKET,SO_BROADCAST,
                    (char *)&optval,optlen) < 0)
          perror("setsockopt(BROADCAST)");
#endif
#ifdef DONTROUTE
     printf("\nExecute SETSOCKOPT to disallow packet routing\n");
     optval = 1;
```

```
     optlen = sizeof(optval);
     if (setsockopt(channel,SOL_SOCKET,SO_DONTROUTE,
                    (char *)&optval,optlen) < 0)
          perror("setsockopt(DONTROUTE)");
#endif
#ifdef SETBUF
     printf("\nExecute SETSOCKOPT to increase socket buffering\n");
     optlen = sizeof(optval);
      optval = 10*1024;
     if (setsockopt(channel,SOL_SOCKET,SO_RCVBUF,
                    (char *)&optval,optlen) < 0)
          perror("setsockopt(RCVBUF)");
     optlen = sizeof(optval);

     optval = 10*1024;
     if (setsockopt(channel,SOL_SOCKET,SO_SNDBUF,
                    (char *)&optval,optlen) < 0)
          perror("setsockopt(RCVBUF)");
#endif


     printf("\nExecute GETSOCKOPT to determine socket options\n");
     optlen = sizeof(optval);
     if (getsockopt(channel,SOL_SOCKET,SO_BROADCAST,
                    (char *)&optval,&optlen) < 0)
          perror("getsockopt(BROADCAST)");

     else

          printf("   Broadcast mode is turned %s\n",optval ? "ON" : "OFF");
     optlen = sizeof(optval);
     if (getsockopt(channel,SOL_SOCKET,SO_DONTROUTE,

                     (char *)&optval,&optlen) < 0)
          perror("getsockopt(DONTROUTE)");
     else
          printf("   Dontroute mode is turned %s\n",optval ? "ON" : "OFF");

     optlen = sizeof(optval);
     if (getsockopt(channel,SOL_SOCKET,SO_RCVBUF,
                    (char *)&optval,&optlen) < 0)
          perror("getsockopt(RCVBUF)");
     else
          printf("   Receive buffer size is %d bytes\n",optval);
     optlen = sizeof(optval);
     if (getsockopt(channel,SOL_SOCKET,SO_SNDBUF,
                    (char *)&optval,&optlen) < 0)
          perror("getsockopt(SNDBUF)");
     else
          printf("   Send buffer size is %d bytes\n",optval);
#ifdef CONNECTIONLESS
     printf("\nUsing CONNECTIONLESS version...\n");
#else
     printf("\nUsing CONNECTED version...\n");

     len = sizeof(remote);
     if (connect(channel, &remote, len) < 0) {
          perror ("connect");
          exit (0);
     }
#endif
     printf("\nExecute GETSOCKNAME to determine my socket's address and \
port\nAddress is zero if CONNECTIONLESS\n");
     optlen = sizeof(me);
     if (getsockname(channel,(struct sockaddr *)&me,&optlen) < 0)
          perror("getsockname");
     else
          printf("My socket: family=%d port=%d addr=%lx\n",
               me.sin_family,me.sin_port,me.sin_addr.s_addr);

     /*
      * Write it over the network
```

```
      */

     buffer[0] = '?';
     while (buffer[0] != '!') {
          int sent = 0;
          printf("\nInput (end with !)? ");
          if (gets(buffer) == NULL) break;
          if (buffer[0] == 0) continue;
          tosend = (int)strlen(buffer);

retry:
          printf("\nExecute SEND[TO]\n");

#ifdef CONNECTIONLESS
          len = sizeof(remote);
          status = sendto(channel, ((char *)buffer + sent), tosend, 0,
                    (struct sockaddr *)&remote, len);
#else
          status = send(channel, ((char *)buffer + sent), tosend, 0);
#endif
          printf("\nAfter SEND[TO], execute GETSOCKNAME\n");

          optlen = sizeof(me);
          if (getsockname(channel,(struct sockaddr *)&me,&optlen) < 0)
               perror("getsockname");
          else
               printf("After send, my socket: family=%d port=%d addr=%lx\n",
                    me.sin_family,me.sin_port,me.sin_addr.s_addr);
          switch (status) {
               case 0:
                    DELAY(5L);
                    goto retry;
               case -1:
                    perror ("echo client: send failed");
                    break;
               default:
                    if ( (sent = sent + status) < tosend) {
                         tosend = tosend - sent;
                         goto retry;
                    }
                    break;
          }

          /*
           * Read from the network
           */

          printf("\nExecute SOCKET_IOCTL to determine chars on read queue");
          if (socket_ioctl (channel, FIONREAD, (char *)&ncc) < 0) {
               perror ("socket_ioctl(FIONREAD)");
               ncc = 1;
          }
          else
               printf("   Socket_ioctl(FIONREAD) returns %d chars\n", ncc);
          while (ncc) {
               len = sizeof(him);
               tosend = sizeof(buffer);
               printf("\nExecute RECV[FROM]\n");
#ifdef CONNECTIONLESS
               status = recvfrom(channel, (char *)&buffer[0], tosend,
                    0, (struct sockaddr *)&him, &len);
#else
               status = recv(channel, (char *)&buffer[0], tosend, 0);
#endif

               if(status == -1)
                    perror ("echo client: receive failed");
               else {
                    buffer[status] = 0;
#ifdef CONNECTIONLESS
           printf("After RECVFROM, his socket: family=%d port=%d addr=%lx\n",
                       him.sin_family,him.sin_port,him.sin_addr.s_addr);
```

```
#endif
          printf("Number of chars from recv[from] is %d\n",
                       status);
          }
          printf("\nExecute GETPEERNAME fails if CONNECTIONLESS socket\n");
           optlen = sizeof(him);
          if (getpeername(channel,(struct sockaddr *)&him,&optlen) < 0)
                   perror("getpeername");
          else
              printf("   His socket: family=%d port=%d addr=%lx\n",
                        him.sin_family,him.sin_port,him.sin_addr.s_addr);
          printf("\n   Data from net: %s\n", buffer);
          printf("\nExecute SOCKET_IOCTL to determine chars on re queue\n") ;
          ncc = 0;
          if (socket_ioctl (channel, FIONREAD, (char *)&ncc) < 0)
                   perror ("socket_ioctl(FIONREAD)");
          else
                   printf("   Socket_ioctl(FIONREAD) returns %d chars\n", ncc);
       }
    }

    /*
     *  Close socket
     */

    printf("\nExecute SHUTDOWN to close socket\n");
    if (shutdown(channel,2) < 0)
         perror("shutdown");

}
```

UDP Server Program

The following programming example shows how to use the socket routines in a server application:

```
#pragma nolist
#include <$system.ztcpip.param.h>
#include <$system.ztcpip.socket.h>
#include <$system.ztcpip.in.h>
#include <$system.ztcpip.netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>
#include <cextdecs(DELAY)>
#pragma list

#define PORT_ECHO 1987
int chan;
struct sockaddr_in sin, remote;
int len;
char buf[10*1024];

main ()
{
        int status;
        int optval, optlen;

        /*
         * Set your local address
         */

        sin.sin_port = htons(PORT_ECHO); /* Interchange bytes of PORT  */
        sin.sin_addr.s_addr = INADDR_ANY;
        sin.sin_family = AF_INET;

        /*
         * Create a socket
         */
```

```
        chan = socket(AF_INET, SOCK_DGRAM, 0);
        if (chan == -1){
                printf ("echo server: socket failed\n");
                exit (0);
        }

        /*
         * Bind it to an Internet Address
         */

        len = sizeof(sin);
        status = bind(chan, (struct sockaddr *)&sin, len);
        if (status == -1)
                perror ("echo server: bind failed");
        optlen = sizeof(optval);
        optval = 1;
        if (setsockopt(chan,SOL_SOCKET,SO_BROADCAST,
                        (char *)&optval,optlen) < 0)
                perror("setsockopt");
        optlen = sizeof(optval);
        optval = 20*1024;
        if (setsockopt(chan,SOL_SOCKET,SO_RCVBUF,
                        (char *)&optval,optlen) < 0)
                perror("setsockopt(RCVBUF)");
        optlen = sizeof(optval);
        optval = 20*1024;
        if (setsockopt(chan,SOL_SOCKET,SO_SNDBUF,
                        (char *)&optval,optlen) < 0)
                perror("setsockopt(SNDBUF)");
        while (1)
        {
                int tosend, sent = 0;
                len = sizeof(remote);
                tosend = sizeof( buf);
                status = recvfrom(chan, (char *)&buf[0], tosend, 0,
                        (struct sockaddr *)&remote, &len);
                if (status == -1)
                        perror ("echo server: recvfrom failed");
                else
                        buf[status] = 0;
                if (buf[0] == 0) continue;
                tosend = (int)strlen(buf);

retry:
                len = sizeof(remote);
                status = sendto(chan, ((char *)buf + sent), tosend, 0,
                                (struct sockaddr *)&remote, len);
                switch (status) {
                        case 0:
                                DELAY(5L);
                                goto retry;
                        case -1:
                                perror ("echo server: send failed");
                                break;
                        default:
                                if ( (sent = sent + status) < tosend) {
                                        tosend = tosend - sent;
                                        goto retry;
                                }
                                break;

                }
        }
}
```

UDP Program for Sending Multicast Packets

The following programming example shows how to use the socket routines in an application that implements multicast for sending:

```
/*#pragma nolist*/
#include "inh"
#include "socketh"
#include "sckconfh"
#include <errnoh>
#include <routeh>
#include <paramh>

#include <ioctlh>
#include <stdioh>
#include <stringh>
#include <memoryh>
#include <stdlibh>
#include <cextdecs(DEBUG,FILE_GETINFO_,AWAITIOX,SETMODE,DELAY)>
#include <fcntlh>
#include <ctypeh>
#include <timeh>
#pragma list

#define BUFFER_LEN      10000
#define PORT_LEN        4
#define HOST_LEN        4
#define MAGIC_NUMBER    0x00D71101L

int main (int argc, char **argv)
{
  struct protoent     *udproto;
  struct sockaddr_in  sin, this, to;
  struct hostent      *temp;
  struct in_addr      in_addr_gmulti, in_addr_multi0, in_addr_mult;
  struct in_addr      in_addr_this;
  struct ip_mreq      multi_req;


  int    x, i, j, k, fd1, req_count = BUFFER_LEN , xcount, loopCount;
  int    len, tolen;
  int    portNum = 0, argNum = 1, bytesready, error;
  int    getsize, ssockerr = 0;
  long   dtime;
  time_t timenow;

  FILE   *fi;

  char   hostchar[HOST_LEN+1];
  char   ttlset, ttlget, loopbkset, loopbkget;
  char   *chr = "-", *multiip, *ascptr, *thishost, *thisip;

  char   sendbuf[BUFFER_LEN];

  unsigned long   thisaddr, multiaddr, multiaddr0;


  if (argc != 10)  {
      printf("usage: sndmulw [NO]DEBUG  tcpip_process  port  this_host");
      printf("  multicast_ip  ttl  loopCount  data_file  send_size\n");
      exit (0);
  }

  if (!strcmp (argv[argNum++],"DEBUG"))
      DEBUG();

  /* TCPIP^PROCESS^NAME parameter */

  printf ("\nClient Process: %s\n", argv[argNum]);
  socket_set_inet_name (argv[argNum++]);

  /* Port number */
```

```
portNum = atoi (argv[argNum++]);        /* convert string to PORT # */
printf ("  PortNum: %i\n", portNum);

/* Name of this host */

thishost = argv[argNum++];

if ((temp = gethostbyname (thishost)) != (struct hostent*)NULL)  {
    memmove ((char *)&in_addr_this.s_addr, (char *)temp->h_addr,
             (size_t)temp->h_length);
}
else {
    printf ("gethostbyname failed for %s, error = %d\n", thishost, h_errno);
    exit (0);
}

thisaddr = in_addr_this.s_addr;
thisip = inet_ntoa (in_addr_this);
printf ("Multicast Interface IP: %s\n", thisip);

/* IP address of the multicast group to join */

multiip = argv[argNum++];
multiaddr0 = inet_addr (multiip);           /* convert to binary format   */

/* Multicast TTL */

ttlset = atoi(argv[argNum++]);
printf ("Multicast TTL: %i\n",ttlset);

/* Test loop count */

loopCount = atoi (argv[argNum++]);

/* Protocol is UDP */

udproto = getprotobyname ("UDP");

/* Open data input file */

if ((fi = fopen (argv[argNum++],"r")) == NULL) {
    printf ("OPEN failed for the data input file\n");
    exit (0);
}


req_count = atoi (argv[argNum++]);

if (req_count > BUFFER_LEN)

    req_count = BUFFER_LEN;
printf ("Requested count : %i\n",req_count);

xcount = fread (sendbuf,req_count,1,fi);
if (xcount != 1)  {
    printf ("Error reading Input file.  Check if it's in subvol!\n");
    exit (0);
}

sendbuf[req_count-1] = '\0';

/* Create socket */

if ((fd1 = socket (AF_INET, SOCK_DGRAM, udproto->p_proto)) < 0){
    perror ("Socket Failure");
    exit (0);
}

/* Test Multicast I/F set and get */

printf ("SETting Multicast I/F to %s or 0x%lx \n", thisip, thisaddr);
```

```
 if (setsockopt (fd1, IPPROTO_IP, IP_MULTICAST_IF,
                 (char *)&in_addr_this, sizeof(in_addr_this)))  {
     perror ("SET MULTI IF error");
     exit (0);
 }

 if (getsockopt (fd1, IPPROTO_IP, IP_MULTICAST_IF,
                 (char *)&in_addr_gmulti, &getsize))  {
     perror ("GET MULTI IF error");
     exit (0);
 }

 printf ("GET Multicast I/F: %s, size: %d\n", inet_ntoa(in_addr_gmulti),
         getsize);

 /* Disable multicast loopback */

 loopbkset = 0;
 if (setsockopt (fd1, IPPROTO_IP, IP_MULTICAST_LOOP,
                 (char *)&loopbkset, sizeof(loopbkset)))  {
     perror ("SET MULTI LOOP error");
     exit (0);
 }
 printf ("Multicast loopback is disabled\n");

 /* Set multicast TTL */

 ttlget = 0;
 printf ("SETting TTL to %d\n",ttlset);
 if (setsockopt (fd1, IPPROTO_IP, IP_MULTICAST_TTL,
                 (char *)&ttlset, sizeof(ttlset)))
     perror("SET MULTI TTL error");
 if (getsockopt (fd1, IPPROTO_IP, IP_MULTICAST_TTL,
                 (char *)&ttlget, &getsize))
     perror("GET MULTI TTL error");
 printf ("GET TTL: %d, size: %d \n",ttlget, getsize);

***************/
 /* Send data to the multicast groups */

 to.sin_family = AF_INET;
 to.sin_port = portNum;
 tolen = sizeof(to);
 srand((unsigned int) timenow);    /* initialize random number gen */

 for (i = 0; i < loopCount; i++) {
     printf ("Loop: %d\n", i+1);
     for (j = 0, multiaddr = multiaddr0; (j < IP_MAX_MEMBERSHIPS);
         j++, multiaddr += MAGIC_NUMBER)  {
         to.sin_addr.s_addr = multiaddr;
         ascptr = inet_ntoa (to.sin_addr);
         for (k = 0; *ascptr != 0; k++)
             sendbuf[k] = *ascptr++;
         for (;k < 15; k++)
             sendbuf[k] = *chr;
         timenow = time(NULL);
         if ((xcount = sendto (fd1, sendbuf, req_count, 0,
                               (struct sockaddr *)&to, tolen)) < 0)  {
             perror ("  Sendto failure");
             exit (0);
         }
         else
             printf ("%s SENDTO completed %i bytes to: %s\n", ctime(&timenow),
                      xcount, inet_ntoa(to.sin_addr));
         dtime = (rand() % 150) + 50L; /* 0.5 - 2 seconds */
         DELAY (dtime);
     } /* end for j loop */
     dtime = 100L; /* 1 second */
     DELAY (dtime);
 }  /* end for i loop */
```

```
  /* Close the socket */
  FILE_CLOSE_ (fd1);
}
```

UDP Program for Receiving Multicast Packets

The following programming example shows how to use the socket routines in an application that implements multicast for receiving:

```
#pragma nolist
#include "inh"
#include "sckconfh"
#include "socketh"
#include <errnoh>
#include <routeh>
#include <paramh>
#include <ioctlh>
#include <stdioh>
#include <stringh>
#include <memoryh>
#include <stdlibh>
#include <cextdecs(DEBUG,FILE_GETINFO_,AWAITIOX,SETMODE)>
#include <fcntlh>
#include <ctypeh>
#pragma list

#define BUFFER_LEN      10000
#define PORT_LEN        4
#define HOST_LEN        4
#define DELAYTIME       200
#define MAGIC_NUMBER    0x00D71101L

int main (int argc, char **argv)
{
  struct protoent     *udproto;
  struct sockaddr_in  sin, this, to, from;
  struct hostent      *temp;
  struct in_addr      in_addr_gmulti, in_addr_multi0, in_addr_mult;
  struct in_addr      in_addr_this;
  struct ip_mreq      multi_req;

  int    x, i, j, k, fd1, req_count, xcount;
  int    len, fromlen;
  int    portNum, argNum = 1, error;
  int    getsize, ssockerr = 0;

  FILE   *fi;

  char   hostchar[HOST_LEN+1];
  char   ttlset, ttlget, loopbkset, loopbkget;
  char   *multiip, *ascptr, *thishost, *thisip;

  char   recvbuf[BUFFER_LEN];

  unsigned long   thisaddr, multiaddr, multiaddr0;


  if (argc != 7)  {
    printf("usage: rcvmcl  [NO]DEBUG  tcpip_process  port  this_host");
    printf("  multicast_ip  ttl\n");
    exit (0);
  }

  if (!strcmp (argv[argNum++],"DEBUG"))
    DEBUG();

  /* TCPIP^PROCESS^NAME parameter */

  printf ("\nClient Process: %s\n", argv[argNum]);
  socket_set_inet_name (argv[argNum++]);
```

```
/* Port number */


portNum = atoi (argv[argNum++]);        /* convert string to PORT # */
printf ("  PortNum: %i\n", portNum);

/* Name of this host */

thishost = argv[argNum++];
if ((temp = gethostbyname (thishost)) != (struct hostent*)NULL)  {
  memmove ((char *)&in_addr_this.s_addr, (char *)temp->h_addr,
           (size_t)temp->h_length);
}
else {
  printf ("gethostbyname failed for %s, error = %d\n", thishost, h_errno);
  exit (0);
}

thisaddr = in_addr_this.s_addr;
thisip = inet_ntoa (in_addr_this);
printf ("Multicast Interface IP: %s\n", thisip);



/* IP address of the multicast group to join */

multiip = argv[argNum++];
multiaddr0 = inet_addr (multiip);           /* convert to binary format   */

/* Multicast TTL */

ttlset = atoi(argv[argNum++]);
printf ("Multicast TTL: %i\n",ttlset);

/* Protocol is UDP */
udproto = getprotobyname ("UDP");

/* Create socket */


if ((fd1 = socket (AF_INET, SOCK_DGRAM, udproto->p_proto)) < 0){
  perror ("Socket Failure");
  exit (0);
}

/* Test Multicast I/F set and get */

printf ("SETting Multicast I/F to %s or 0x%lx \n", thisip, thisaddr);

if (setsockopt (fd1, IPPROTO_IP, IP_MULTICAST_IF,
                (char *)&in_addr_this, sizeof(in_addr_this)))  {
  perror ("SET MULTI IF error");
  exit (0);
}

if (getsockopt (fd1, IPPROTO_IP, IP_MULTICAST_IF,
                (char *)&in_addr_gmulti, &getsize))  {
  perror ("GET MULTI IF error");
  exit (0);
}

printf ("GET Multicast I/F: %s, size: %d\n", inet_ntoa(in_addr_gmulti),
        getsize);

/* Set multicast TTL */

ttlget = 0;
printf ("SETting TTL to %d\n",ttlset);
if (setsockopt (fd1, IPPROTO_IP, IP_MULTICAST_TTL,
                (char *)&ttlset, sizeof(ttlset)))
  perror("SET MULTI TTL error");
if (getsockopt (fd1, IPPROTO_IP, IP_MULTICAST_TTL,
```

```
                (char *)&ttlget, &getsize))
  perror("GET MULTI TTL error");
printf ("GET TTL: %d, size: %d \n",ttlget, getsize);

/* Join multicast groups */

multi_req.imr_interface.s_addr = thisaddr;
for (i = 1, multiaddr = multiaddr0; i <= IP_MAX_MEMBERSHIPS;
     i++, multiaddr += MAGIC_NUMBER)  {
  multi_req.imr_multiaddr.s_addr = multiaddr;
  printf ("ADDing MEMBERSHIP to group: %s or %lx\n",

          inet_ntoa (multi_req.imr_multiaddr) ,
          multi_req.imr_multiaddr.s_addr);
  printf (" ON I/F: %s\n", inet_ntoa(multi_req.imr_interface));
  if (setsockopt (fd1, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                  (char *)&multi_req, sizeof(multi_req)))  {
    perror ("ADD MEMBER error");
    printf ("           error code: %x Hex (%d.)\n", errno, errno);
  }
}

/* Bind */

sin.sin_family = AF_INET;
sin.sin_port = portNum;
sin.sin_addr.s_addr = INADDR_ANY;
len = sizeof(sin);


if (bind (fd1, (struct sockaddr *)&sin, len) < 0) {
  perror ("Bind Failure");
  exit (0);
}

/* Receive from multicast */

fromlen = sizeof(from);

i = 1;
while (1)  {                    /* standby for receiving always */
  printf ("\n\n.............................\n");
  printf (".............................\n");
  printf (".............................\n");
  printf (".............................\n");
  printf ("... LOOP %d\n", i);
  printf (".............................\n");
  printf (".............................\n\n\n");

                                     /* For every 10 loop, add some      */
  if ((i % 10) == 0)  {              /* memberships                      */
    printf ("ADD every other 3 group memberships\n");
    for (j = 1, multiaddr = multiaddr0;
         j <= IP_MAX_MEMBERSHIPS;
         j += 3, multiaddr += (MAGIC_NUMBER * 3))  {
      multi_req.imr_multiaddr.s_addr = multiaddr;
      printf ("ADD MEMBERSHIP to group: %s or %lx\n",
              inet_ntoa (multi_req.imr_multiaddr),
              multi_req.imr_multiaddr.s_addr);
      printf (" ON I/F: %s\n", inet_ntoa(multi_req.imr_interface));
      if (setsockopt (fd1, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                      (char *)&multi_req, sizeof(multi_req)))  {

        perror ("ADD MEMBER error");
        printf ("           error code: %x Hex (%d.)\n", errno, errno);
      }
    }
  }
  else                               /* For every x5 loop, drop some     */
    if ((i % 5) == 0)  {             /* memberships                      */
      printf ("DROP every other 3 group memberships\n");
      for (j = 1, multiaddr = multiaddr0;
```

```
            j <= IP_MAX_MEMBERSHIPS;
            j += 3, multiaddr += (MAGIC_NUMBER * 3))  {
          multi_req.imr_multiaddr.s_addr = multiaddr;
          printf ("DROP MEMBERSHIP from group: %s or %lx\n",
                  inet_ntoa (multi_req.imr_multiaddr),
                  multi_req.imr_multiaddr.s_addr);
          printf (" ON I/F: %s\n", inet_ntoa(multi_req.imr_interface));
          if (setsockopt (fd1, IPPROTO_IP, IP_DROP_MEMBERSHIP,
                          (char *)&multi_req, sizeof(multi_req)))  {
            perror ("DROP MEMBER error");
            printf ("           error code: %x Hex (%d.)\n", errno, errno);
          }
        }
      }
    req_count = 1000 * IP_MAX_MEMBERSHIPS;
    while (req_count)  {
      /* printf ("Retrieving %d bytes\n", req_count); */
      if ((xcount = recvfrom (fd1, recvbuf, req_count, 0,
                              (struct sockaddr *)&from,
                              (int *)&fromlen)) < 0) {
        perror ("  Recvfrom failure");
        exit (0);
      }
      printf ("Loop %d..............received %i bytes from %s\n",
              i, xcount, inet_ntoa (from.sin_addr));
      recvbuf[xcount] = 0;
      recvbuf[72] = 0;      /* to print the first 72 chars only */
      printf ("%s\n",recvbuf);

      req_count -= xcount;
    }
    i++;
  }     /* end for loop */

  close (fd1);
}
```

TAL Echo Client Programming Example

The TAL program below demonstrates an ECHO client that communicates with an ECHO server. The source code for this program appears in the TALDOCUM file on the site update tape (SUT) for TAL sockets. Refer to the *TCP/IP Applications and Utilities User Guide* for details on using ECHO.

```
?ENV COMMON
?SYMBOLS,INSPECT
?SEARCH $SYSTEM.SYSTEM.CLULIB
?SEARCH $SYSTEM.SYSTEM.TALLIB
?SEARCH $SYSTEM.ZTCPIP.libinetl

NAME echo_example;
--
-- This sample TAL socket program communicates with an ECHO server.
--
?PUSHLIST,NOLIST,SOURCE $SYSTEM.SYSTEM.CREDECS(initialization,Termination)
?POPLIST
BLOCK sockdeft;
?PUSHLIST,NOLIST,SOURCE $SYSTEM.ZTCPIP.SOCKDEFT
?POPLIST
END BLOCK;
BLOCK error_codes;
?PUSHLIST,NOLIST, SOURCE $SYSTEM.ZTCPIP.SOCKPROC(error_codes)
?POPLIST
END BLOCK;
BLOCK getsockopt_opts;
?PUSHLIST,NOLIST, SOURCE $SYSTEM.ZTCPIP.SOCKPROC(getsockopt_opts)
?POPLIST
END BLOCK;
BLOCK socket_opts;
?PUSHLIST,NOLIST,SOURCE $SYSTEM.ZTCPIP.SOCKPROC(socket_opts)
?POPLIST
```

```
END BLOCK;
?PUSHLIST,NOLIST,SOURCE $SYSTEM.SYSTEM.RTLDECS(convert)
?POPLIST
?PUSHLIST,NOLIST,SOURCE $SYSTEM.ZTCPIP.SOCKPROC( connect
?                                               ,gethostbyname
?                                               ,gethostbyaddr
?                                               ,getservbyname
?                                               ,get_errno
?                                               ,inet_addr
?                                               ,paramcapture
?                                               ,recv
?                                               ,send
?                                               ,socket
?                                               )
?POPLIST

?PUSHLIST,NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS0(
?                                               DEBUG
?                                               FILE_CLOSE_
?                                             )
?POPLIST
?PUSHLIST,NOLIST,SOURCE $SYSTEM.SYSTEM.CREDECS( cre_terminator_
?                                              ,cre_log_message_
?                                              )
?POPLIST
?PUSHLIST,NOLIST,SOURCE $SYSTEM.SYSTEM.CLUDECS( SMU_Param_GetText_
?                                               SMU_Startup_GetText_
?                                               )
?POPLIST
?PUSHLIST,NOLIST,SOURCE $SYSTEM.SYSTEM.RTLDECS( RTL_STRLENX_
?                                               RTL_Int16_to_decimal_
?                                               )
?POPLIST
?PUSHLIST,NOLIST,SOURCE $SYSTEM.SYSTEM.TALDECS(tal_cre_initializer_)
?POPLIST
-- Heap directive is necessary either in the MAIN program or in
-- the BIND step.  If there is no HEAP directive, then the
-- C Language functions using the heap (malloc, calloc, realloc)
-- fails. The heap directive is put into this program for saftey in a
-- mixed language environment, it is NOT required to make use of the
-- Socket library, which makes no use of HEAP functions for memory
-- management.
?HEAP 20
?EXTENDSTACK 8

PROC term_msg(message);
STRING .EXT message;
BEGIN
INT error := 0;
    IF (error := CRE_LOG_MESSAGE_(message:$INT(RTL_STRLENX_(message))))
    THEN BEGIN
        CALL DEBUG;
    END;
END;

PROC PRINT_ERROR(prefix);
STRING .EXT prefix;
BEGIN
    STRING .EXT     work_buf[0:300];
    STRING .EXT     s := -1D;

    work_buf ':=' prefix FOR $INT(RTL_STRLENX_(prefix))
     & " Error = " -> @s;
    CALL RTL_Int16_to_decimal_(get_errno,s,6,RTL^Leading^separate);
    @s := @s + 6D;
    s := 0; -- Null Termination.
    CALL term_msg(work_buf);
END;

INT PROC term_read(input_buffer:buffer_length);
STRING .EXT input_buffer;
INT(32)     buffer_length;
```

```
BEGIN
INT count_read := 0;
INT error := 0;
    IF (error:=CRE_LOG_MESSAGE_(input_buffer:0,,buffer_length,count_read))
    THEN BEGIN
        CALL DEBUG;

    END;
    input_buffer[count_read] := 0; -- Null Termination.

    RETURN count_read;

END;


PROC echo_main MAIN;
BEGIN
    INT(32)         bytes_from_term := 0;
    INT(32)         total_received := 0;
    INT(32)         nrcvd := 0;
    INT(32)         sock := -1;
    INT(32)         bytes_returned := 0;
    STRING  .EXT    startup_msg[0:50];
    STRING  .EXT    buf[0:1024];
    INT     .EXT    param_msg = buf;
    STRING  .EXT    host_name;
    STRUCT  .EXT    sin(sockaddr_in);
    STRUCT  .EXT    hp(hostent);
    STRUCT  .EXT    se(servent);

    -- All of the following strings are NULL terminated, this is the
    -- convention in C and many of the Socket routines depend on null
    -- terminated strings.

    STRING          echo_service    = 'P'   := ["echo",0];
    STRING          TCP_PROTOCOL    = 'P'   := ["tcp",0];
    STRING          socket_error    = 'P'   := ["Socket error",0];
    STRING          send_error      = 'P'   := ["Send error",0];
    STRING          recv_error      = 'P'   := ["Recv error",0];
    STRING          connect_error   = 'P'   := ["Connect error",0];
    STRING          string_portion  = 'P'   := ["STRING",0];
    STRING          usage           = 'P'   := ["usage: echo machine",0];
    STRING          no_echo_serv    = 'P'
                        := ["Echo Service not defined, check SERVICES file.",0];
    STRING          con_close       = 'P'
                        := ["Connection unexpectedly closed by host.",0];
    STRING          ALL             = 'P'   := ["*ALL*",0];
    INT count := 0;

    -- Initialization uses the facilities of the CRE to
    -- facilitate the possibility of a mixed language environment.

    CALL tal_cre_initializer_(CRE^Save^all^messages);

    -- Use SMU routines to read the startup message.
    count := SMU_Startup_GetText_(
                 string_portion:$INT(RTL_STRLENX_(string_portion))
                ,startup_msg:$OCCURS(startup_msg));
    startup_msg[count] := 0;    -- Null Termination.
    -- Display the usage of this program if there was no startup text.
    IF NOT count
    THEN BEGIN
        CALL term_msg(usage);
        CALL CRE_TERMINATOR_(CRE^Completion^normal);
    END;

    -- Use SMU to get ENTIRE parameter message and if there is one
    -- call the paramcapture routine. The paramcapture routine is
    -- necessary to save parameters such as TCPIP^PROCESS^NAME in
    -- socket library data structures.
    IF (SMU_Param_GetText_(  ALL:$INT(RTL_STRLENX_(ALL))
                            ,buf:$INT($OCCURS(buf)))) > 0
```

```
THEN BEGIN
    CALL paramcapture(param_msg);
END;


-- Create an open socket to do IO on.
IF  ((sock := socket (AF_INET, SOCK_STREAM, 0)) < 0)
THEN BEGIN
    CALL PRINT_ERROR(socket_error);
    CALL CRE_TERMINATOR_(CRE^Completion^fatal);
END;

-- Look up the port number of the echo service using a socket
-- routine  (echo port is well known port 7)
IF (@se := getservbyname(echo_service,TCP_PROTOCOL)) = 0D
THEN BEGIN
    term_msg(no_echo_serv);
    CALL CRE_TERMINATOR_(CRE^Completion^warning);
END;

-- Start filling up the sockaddr_in structure for a connect.
sin.sin_port := se.s_port;  -- From getservbyname
sin.sin_family := AF_INET;

-- Check to see if address was supplied in dotted decimal format.
IF (sin.sin_addr.s_addr := inet_addr(startup_msg)) = -1D
THEN BEGIN
    -- It is not dotted decimal, check to see if it can be resolved
    -- in a name lookup.
    @hp := gethostbyname(startup_msg);
    IF (@hp = 0D)
    THEN BEGIN
        buf ':=' "Unknown host: "
         & startup_msg FOR $INT(RTL_STRLENX_(startup_msg))
         & 0; -- Null Termination.
        CALL term_msg(buf);
        CALL CRE_TERMINATOR_(CRE^Completion^warning);
    END;
    sin.sin_addr.s_addr ':=' hp.h_addr_list.ptrs FOR hp.h_length;
    @host_name :=@hp.h_name;
END ELSE BEGIN
    @hp := gethostbyaddr (sin.sin_addr.s_addr, 4, AF_INET);
    if (@hp = 0D)
    THEN BEGIN
        @host_name := @startup_msg;
    END ELSE BEGIN
        @host_name := @hp.h_name;
    END;

END;
buf ':=' "Establishing Connection to: "
 & host_name FOR $INT(RTL_STRLENX_(host_name))
 & 0;  -- Null Termination.
CALL term_msg(buf);
IF (connect(sock,sin,$LEN(sin)) < 0)
THEN BEGIN
    CALL PRINT_ERROR(connect_error);
    CALL CRE_TERMINATOR_(CRE^Completion^fatal);
END;
buf ':=' "Connected" & 0;
CALL term_msg(buf);
WHILE (bytes_from_term := term_read(buf:$OCCURS(buf))) > 0
DO BEGIN
    IF (send(sock,buf,bytes_from_term,0)) <= 0
    THEN BEGIN
        CALL PRINT_ERROR(send_error);
        CALL CRE_TERMINATOR_(CRE^Completion^fatal);
    END;

    -- Use the following loop because the socket interface may
    -- require more than one call to "recv" to get all of the
    -- bytes desired. This is usually due to network fragmentation.
```

```
        total_received := 0;
        DO BEGIN
            nrcvd := 0;

            IF ((nrcvd := recv(  sock

                               ,buf[total_received]
                               ,$OCCURS(buf)-total_received
                               ,0)) < 0)
            THEN BEGIN
                PRINT_ERROR(recv_error);
                CALL CRE_TERMINATOR_(CRE^Completion^fatal);
            END;
            IF (nrcvd = 0)
            THEN BEGIN
                term_msg(con_close);
                CALL CRE_TERMINATOR_(CRE^Completion^warning);
            END;
            total_received := total_received + nrcvd;
        END UNTIL total_received >= bytes_from_term;
        buf[total_received] := 0;   -- Null Termination.
        CALL term_msg(buf);
    END;

    CALL FILE_CLOSE_(sock);
    CALL CRE_TERMINATOR_(CRE^Completion^normal);
END;
```

Using AF_INET6 Sockets

This section contains a client and server program that use AF_INET6 sockets.

AF_INET6 Client Stub Routine

This example shows a sample client program that you can build, compile, and run on your system. The program sends a request to and receives a response from the system specified on the command line. All addresses are in IPv6 address format.

```
/*
 *      AF_INET6 Client Stub Routine
 * ****************************************************************
 * *                                                              *
 * *    Copyright (c) Hewlett-Packard Company, 2003            *
 * *                                                              *
 * *   The software contained on this media  is  proprietary  to  *
 * *   and  embodies  the  confidential  technology  of  Hewlett  *
 * *   Packard Corporation.  Possession, use,  duplication  or    *
 * *   dissemination of the software and media is authorized only *
 * *   pursuant to a valid written license from Hewlett Packard   *
 * *   Corporation.                                               *
 * *                                                              *
 * *   RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure *
 * *   by the U.S. Government is subject to restrictions  as  set *
 * *   forth in Subparagraph (c)(1)(ii)  of  DFARS  252.227-7013, *
 * *   or  in  FAR 52.227-19, as applicable.                      *
 * *                                                              *
 * ****************************************************************
 */
#include <systype.h>
#include <socket.h>
#include <errno.h>
#include <in.h>

#include <in6.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
```

```
#include <signal.h>
#include <stdlib.h>
#include <inet.h>
#include <nameser.h>
#include <cextdecs(FILE_CLOSE_)>
#define SERVER_PORT 7639
#define CLIENT_PORT 7739
#define MAXBUFSIZE 4096

int main (
    int argc,
    char **argv )
{
    int             s;
    char            databuf[MAXBUFSIZE];
    int             dcount;
    char            addrbuf[INET6_ADDRSTRLEN];
    char            node[MAXDNAME];
    char            service[MAXDNAME];
    int             ni;
    int             err;
    int             serveraddrlen;
    char            *server;
    struct addrinfo    *server_info;
    struct addrinfo    *cur_info;
    struct addrinfo     hints;

     /* Declare the sockaddr_in6 structure. The use of this type of
        structure is dictated by the communication domain of the
        socket (AF_INET6), which implies communication using the IPv6
        protocol. If you wanted to write a protocol-independent program,
        you would declare a sockaddr_storage structure. */
    struct sockaddr_in6 serveraddr;
    char            request[MAXBUFSIZE] = "This is the client's request";
    if (argc < 2) {
      printf("Usage: client <server>\n");
      exit (0);
    }
    server = argv[1];

    /* Clear the hints structure and set up hints variables. The hints
       structure contains values that direct the getaddrinfo processing.
       In this case, AF_INET6 returns IPv6 addresses. The AI_ADDRCONFIG
       and AI_V4MAPPED values return AAAA records if an IPv6 address is
       configured, and if none are found, return A records if an IPv4
       address is configured. */
    bzero((char *) &hints, sizeof(hints));
    hints.ai_family = AF_INET6;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_ADDRCONFIG | AI_V4MAPPED;
    sprintf(service, "%d", SERVER_PORT);

   /* Obtains the server address. A call to getaddrinfo returns
      IPv6-formatted addresses in one or more structures of type
      addrinfo. */
    err = getaddrinfo(server, service, &hints, &server_info);
    if (err != 0) {
      printf("%s\n", gai_strerror(err));
      if (err == EAI_SYSTEM)
        perror("getaddrinfo");
      exit(2);
    }
    cur_info = server_info;

    /* Create an AF_INET6 socket. The socket type is specified in
```

```
       the addrinfo structure. */
    while (cur_info != NULL) {
      if ((s = socket(cur_info->ai_family,cur_info->ai_socktype,0))<0){
        perror("socket");
        freeaddrinfo(server_info);
        exit(3);
      }

      /* Connect to the server using the address in the addrinfo
         structure named cur_info. */
      if ((err = connect(s,cur_info->ai_addr,(int)cur_info->ai_addrlen))<0){
        perror("connect");
        cur_info = cur_info->ai_next;
        continue;
      }
      break;
    }

    /* Free all addrinfo structures. */
    freeaddrinfo(server_info);
    if (err < 0)
      exit(4);

   /* Send a request to the server. */
    if (send(s, request, (int)strlen(request), 0) < 0) {
      perror("send");
      exit(5);
    }

    /* Receive a response from the server. */
    dcount = recv(s, databuf, sizeof(databuf), 0);
       if (dcount < 0) {
      perror("recv");
      exit(6);
    }
    databuf[dcount] = '\0';
    serveraddrlen = sizeof(serveraddr);

    /* Obtain the address of the peer socket at the other end of the
       connection and store the address in a sockaddr_in6 structure named
       serveraddr. */
    if (getpeername(s, (struct sockaddr*) &serveraddr, &serveraddrlen) < 0){
      perror("getpeername");
      exit(7);
    }
    printf("Response received from");

     /* Obtain the server's name with a call to getnameinfo using the
        address in the sockaddr_in6 structure named serveraddr. The
        NI_NAMEREQD flag directs the routine to return a hostname for the
        given address. */
    ni = getnameinfo((struct sockaddr*)&serveraddr, serveraddrlen,
            node, sizeof(node), NULL, 0, NI_NAMEREQD);
    if (ni == 0)
      printf(" %s", node);
    ni = getnameinfo((struct sockaddr*)&serveraddr, serveraddrlen,
            addrbuf, sizeof(addrbuf), NULL, 0, NI_NUMERICHOST);
    if (ni == 0)
      printf(" (%s)", addrbuf);
    printf(":\n%s\n", databuf);
    FILE_CLOSE_((short)s);
}
```

AF_INET6 Server Stub Program

This example shows a sample server program that you can build, compile, and run on your system. The program receives requests from and sends responses to client programs on other systems.

```
/*
 *      AF_INET6 Server Stub Routine
 * ****************************************************************
 * *                                                              *
 * *    Copyright (c) Hewlett-Packard Company, 2003            *
 * *                                                              *
 * *   The software contained on this media  is  proprietary  to  *
 * *   and  embodies  the  confidential  technology  of  Hewlett  *
 * *   Packard Corporation.  Possession, use,  duplication  or    *
 * *   dissemination of the software and media is authorized only *

 * *   pursuant to a valid written license from Hewlett Packard   *
 * *   Corporation.                                               *
 * *                                                              *
 * *   RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure *
 * *   by the U.S. Government is subject to restrictions  as  set *
 * *   forth in Subparagraph (c)(1)(ii)  of  DFARS  252.227-7013, *
 * *   or  in  FAR 52.227-19, as applicable.                      *
 * *                                                              *
 * ****************************************************************
 */

#include <systypes.h>
#include <socket.h>
#include <errno.h>
#include <in.h>
#include <in6.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <inet.h>
#include <nameser.h>
#include <cextdecs(FILE_CLOSE_)>

#define SERVER_PORT     7639
#define CLIENT_PORT     7739

#define MAXBUFSIZE 4096

int main (
    int argc,
    char **argv )
{
    int             s;
    char            databuf[MAXBUFSIZE];
    int             new_s;
    int             dcount;
    char            addrbuf[INET6_ADDRSTRLEN];
    char            node[MAXDNAME];
    char            port[MAXDNAME];
    int             ni;
    int             clientaddrlen;
   /* Declare the sockaddr_in6 structure named serveraddr. The use
      of this type of structure is dictated by the communication domain
      of the socket (AF_INET6), which implies communication using the IPv6
      protocol. */
    struct sockaddr_in6       serveraddr;
```

```
/* Declare a sockaddr_storage structure named clientaddr. The use
   of this type of structure enables your program to be protocol
   independent. */
sockaddr_storage   clientaddr;
char            response[MAXBUFSIZE] = " This is the server's response";

/* Create an AF_INET6 socket. The socket type SOCK_STREAM is
   specified for TCP or connection-oriented communication. */
if ((s = socket(AF_INET6, SOCK_STREAM, 0)) < 0) {
  perror("socket");
  exit (0);
}

/* Clear the server address and sets up the server variables. */
bzero((char *) &serveraddr, sizeof(struct sockaddr_in6));
serveraddr.sin6_family      = AF_INET6;
serveraddr.sin6_addr        = in6addr_any;

serveraddr.sin6_port        = htons(SERVER_PORT);

/* Bind the server's address to the AF_INET socket. */
if (bind(s, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) < 0){
  perror("bind");
  exit(2);
}

/* Listen on the socket for a connection. The server queues up
   to SOMAXCONN pending connections while it finishes processing the
  previous accept call. See sys_attrs_socket(5) for more information on
  the socket subsystem kernel attributes. */
if (listen(s, SOMAXCONN) < 0) {
  perror("listen");
  FILE_CLOSE_((short)s);
  exit(3);
}
while (1) {
  clientaddrlen = sizeof(clientaddr);

  /* Clear the client address. */
  bzero((char *)&clientaddr, clientaddrlen);

 /* Accept a connection on this socket. The accept call places
    the client's address in the sockaddr_storage structure named
    clientaddr. */
  new_s = accept(s, (struct sockaddr*)&clientaddr, &clientaddrlen);
  if (new_s < 0) {
      perror("accept");
      continue;
  }

  /* Receive data from the client. */
  dcount = recv(new_s, databuf, sizeof(databuf), 0);
  if (dcount < 0) {
      perror("recv");
      FILE_CLOSE_((short)new_s);
      continue;
  }
  databuf[dcount] = '\0';

  printf("Request received from");
  ni = getnameinfo((struct sockaddr *)&clientaddr,
          clientaddrlen, node, sizeof(node), NULL, 0, NI_NAMEREQD);
  if (ni == 0)
    printf(" %s", node);
```

```
      /* Obtains the client's name with a call to getnameinfo using the
         address in the sockaddr_storage structure named clientaddr. The
         NI_NAMEREQD flag directs the routine to return a hostname for
         the given address. */
      ni = getnameinfo((struct sockaddr *)&clientaddr,
              clientaddrlen, addrbuf, sizeof(addrbuf), port, sizeof(port),
              NI_NUMERICHOST|NI_NUMERICSERV);
      if (ni == 0)
        printf(" (%s) port %s", addrbuf, port);
      printf(":\n\"%s\"\n", databuf);

      /* Sends a response to the client. */
      if (send(new_s, response, (int)strlen(response), 0) < 0) {
        perror("send");
        FILE_CLOSE_((short)new_s);
        continue;
      }
      FILE_CLOSE_((short)new_s);
    }
    FILE_CLOSE_((short)s);
}
```

A Well-Known IP Protocol Numbers

Table 17 provides a list of commonly used IP protocol numbers, together with the names you can use for them in your application programs. These protocols are provided in the file `$SYSTEM.ZTCPIP.PROTOCOL`. For other protocol numbers, refer to RFC 1010, "Assigned Numbers."

Table 17 Commonly Used IP Protocol Numbers

Protocol Number	C Name	Protocol	Full Name
0	ip	IP	Internet Protocol (pseudoprotocol number)
1	icmp	ICMP	Internet Control Message Protocol
3	ggp	GGP	Gateway-to-Gateway Protocol
6	tcp	TCP	Transmission Control Protocol
12	pup	PUP	PARC Universal Packet Protocol
17	udp	UDP	User Datagram Protocol

TCP and UDP Port Numbers

Table 18 (page 241), Table 19 (page 242), and Table 20 (page 242) list the port numbers preassigned to specific services when accessed from TCP or UDP. The tables give the name or names of each service as it is used in application programs. These port numbers are provided in the file `$SYSTEM.ZTCPIP.SERVICES`.

Table 18 Port Numbers for Network Services

Port Number	Protocol	C Name(s) of Service or Function
7	TCP, UDP	echo
9	UDP	discard, sink null
11	TCP	systat
13	TCP	daytime
15	TCP	netstat
20	TCP	ftp-data
21	TCP	ftp
23	TCP	telnet
25	TCP	smtp, mail
37	TCP, UDP	time, time server
42	UDP	name, nameserver
43	TCP	whois, nickname (usually to sri-nic)
53	TCP, UDP	domain
101	TCP	hostnames, hostname (usually to sri-nic)
111	TCP, UDP	sunrpc

Table 19 Port Numbers for Host-Specific Functions

Port Number	Protocol	C Name(s) of Service or Function
69	UDP	tftp
77	TCP	rje
79	TCP	finger
87	TCP	link, ttylink
95	TCP	supdup
105	TCP	csnet-ns
117	TCP	uucp-path
119	TCP	nntp, usenet
123	UDP	ntp
1524	TCP	ingreslock

Table 20 Port Numbers for UNIX-Specific Services

Port Number	Protocol	C Name(s) of Service or Function
512	TCP	exec
	UDP	biff, comsat
513	TCP	login
	UDP	who, whod
514	TCP	shell, cmd (no passwords used)
	UDP	syslog
515	TCP	printer, spooler (experimental)
517	UDP	talk
520	UDP	route, router, routed
530	TCP	courier, rpc (experimental)
550	UDP	new-rwho, new-who (experimental)
560	UDP	rmonitor, rmonitord (experimental)
561	UDP	monitor (experimental)

B Socket Errors

This appendix summarizes the socket errors that can be returned in the external variable *errno* by the routines in the socket interface library.

Socket errors start at base 4000.

The errors returned in the external variable *h_errno* are not contained in this appendix. For those errors, see the error descriptions under the gethostbyaddr and gethostbyname functions in Chapter 4 (page 81).

The descriptions given here are general; you should interpret each error according to the type and circumstances of the call. For specific information about the meaning of an error for a particular socket routine, see the description of the individual routine in Chapter 4 (page 81).

Some of the errors defined in $SYSTEM.ZTCPIP.PARAMH are for HP internal use only and cannot be received by application programs using the socket calls. This appendix lists only those socket errors that can be received by application programs.

File-system errors can also be returned in *errno* upon return from a socket call. File-system errors indicate that an error occurred during interprocess I/O. For descriptions of the file-system errors, refer to the *Guardian Procedure Errors and Messages Manual*.

The SAP library function calls described in Chapter 4 (page 81), return file-system errors. For descriptions of the file-system errors, refer to the *Guardian Procedure Errors and Messages Manual*.

The socket errors are described in alphabetical order. The error number associated with each error is shown in parentheses following the mnemonic name of the error. Table 21 (page 253) lists of the errors in numerical order.

Error number definitions can be found in the file $SYSTEM.SYSTEM.ERRNOH.

EACCES (4013)

EACCES

Cause

A call to bind or bind_nw specified an address or port number that cannot be assigned to a nonprivileged user. Only applications whose process access ID is in the SUPER group (user ID 255,n) can bind a socket to a well-known port. 2. The requested operation specified a broadcast address as the destination but the SO_BROADCAST socket option was not enabled (see setsockopt, setsockopt_nw (page 184)).

Effect

The bind, bind_nw, sendto, or sendto_nw call failed.

Recovery

For bind and bind_nw, specify another port number or address, or rerun the application with a process access ID in the SUPER group (user ID 255,n). For sendto or sendto_nw, set the SO_BROADCAST option for the socket.

EADDRINUSE (4114)

EADDRINUSE

Cause

A call to bind or bind_nw specified an address-port number combination that is already in use.

Effect

The bind or bind_nw call failed.

Recovery

Specify another address and port number.

EADDRNOTAVAIL (4115)

EADDRNOTAVAIL

Cause

A call to `bind` or `bind_nw` specified an address-port number combination that is not available on the local host.

Effect

The `bind` or `bind_nw` call failed.

Recovery

Specify an address and port number that are valid for this system.

EAFNOSUPPORT (4113)

EAFNOSUPPORT

Cause

The “Family” attribute in the PROVIDER object is not configured correctly. The PROVIDER object represents a transport-service provider and is associated with the CIPSAM process, which directs socket requests to a specific CLIM. If the attribute is set to "INET", only NonStop TCP/IP is supported. If the attribute is set to "DUAL", both NonStop TCP/IP and NonStop TCP/IPv6 are supported.

Effect

The `socket` or `socket_nw` call failed.

Recovery

For NonStop TCP/IP, specify *`address_family`* as AF_INET.

For NonStop TCP/IPv6, specify *`address_family`* as AF_DUAL.

EALREADY (4103)

EALREADY

Cause

An operation is already in progress. For `accept_nw` and `connect_nw` calls, there is already an outstanding call on the socket. For the `send_nw` call, the send buffer is already full (see the `SO_SNDBUF` option of the setsockopt, setsockopt_nw (page 184) call for increasing the size of the send buffer).

Effect

The call failed.

Recovery

Wait for the operation to complete and retry the call.

EBADF (4009)

EBADF

Cause

The filedes or socket parameter specified in the call contained an invalid file descriptor.

Effect

The call failed.

Recovery

Correct the file descriptor specification in the call and retry the call.

EBADSYS (4196)

EBADSYS

Cause

Either an application attempted to write directly to the NonStop TCP/IPv6 or NonStop TCP/IP process, or an internal error occurred in one of the socket routines.

Effect

The operation failed.

Recovery

Direct writes to the NonStop TCP/IP or NonStop TCP/IP process are not permitted; use the socket calls. However, if the problem appears to be an internal socket error, contact your service provider.

ECONNABORTED (4119)

`ECONNABORTED`

Cause

A connection was aborted by the internal software on your host machine.

Effect

The connection was closed.

Recovery

Close the socket. Reestablish the connection using the `socket`, `bind`, and `connect` calls. If the problem persists, contact your service provider.

ECONNREFUSED (4127)

`ECONNREFUSED`

Cause

The remote host rejected the connection request. This error usually results from an attempt to connect to a service that is inactive on the remote host.

Effect

The `connect` call failed.

Recovery

Start the server on the remote host. Close the local socket. Reestablish the connection using the `socket`, `bind`, and `connect` calls.

ECONNRESET (4120)

`ECONNRESET`

Cause

The peer process reset the connection before the operation completed.

Effect

The `connect` call failed.

Recovery

Close the local socket. Reestablish the connection using the `socket`, `bind`, and `connect` calls.

EDESTADDRREQ (4105)

`EDESTADDRREQ`

Cause

Destination address required. A required address was omitted from an operation on a transport end point.

Effect

The call failed.

Recovery

Retry the call with a valid destination address.

EEXIST (4017)

`EEXIST`

Cause

Object exists. An existing object was specified in an inappropriate context, such as attempting to add a route entry that had already been added.

Effect

The call failed.

Recovery

Retry the call with a valid object name.

EFAULT (4014)

`EFAULT`

Cause

The system encountered a memory access fault in attempting to use an argument of the call.

Effect

The call failed.

Recovery

Contact your service provider.

EHAVEOOB (4195)

`EHAVEOOB`

Cause

Out-of-band data is pending. Before receiving or sending normal data, you must clear the out-of-band data by calling `recv` with the `MSG_OOB` flag set.

Effect

The call failed.

Recovery

Call `recv` with the `MSG_OOB` flag set to read the out-of-band data.

EHOSTDOWN (4128)

`EHOSTDOWN`

Cause

The destination host is present, but it is not responding.

Effect

The call failed.

Recovery

Correct the problem in the destination host and retry the call.

EHOSTUNREACH (4129)

`EHOSTUNREACH`

Cause

No route to host. A transport provider operation was attempted to an unreachable host.

Effect

The call failed.

Recovery

Ensure that you have specified a valid hostname or address. If so, ensure that the remote host can be reached from the local host.

EINPROGRESS (4102)

`EINPROGRESS`

Cause

Operation now in progress. A `connect_nw` call was attempted on a non-blocking socket where `connect_nw` had already been called on that socket.

Effect

The call failed.

Recovery

Wait and retry the operation.

EINTR (4004)

`EINTR`

Cause

While a process was in the sleep mode waiting for an event, it received an unexpected signal, not the wait-for event.

Effect

The call failed.

Recovery

Retry the call.

EINVAL (4022)

`EINVAL`

Cause

The specified socket was already bound to an address or the `address_len` was incorrect.

Effect

The call failed.

Recovery

Corrective action depends on the function and the circumstances. For a list of valid arguments, see the description of the function that caused the error.

EIO (4005)

`EIO`

Cause

I/O error. Some physical I/O error has occurred. In some cases, this error may occur on a call following the one to which it actually applies.

Effect

The call failed.

Recovery

Examine the preceding calls. Retry the call.

EISCONN (4122)

`EISCONN`

Cause

A call to `sendto`, `t_sendto_nw`, `recvfrom`, `recvfrom_nw`, or `t_recvfrom_nw` was made on a socket that was connected.

Effect

The call failed.

Recovery

Correct the call. For a connected socket, use `send`, `send_nw`, *`recv`*, or `recv_nw`.

EMFILE (4024)

`EMFILE`

Cause

The network manager attempted to add too many routes.

Effect

The call failed.

Recovery

Close some files and retry the call.

EMSGSIZE (4106)

`EMSGSIZE`

Cause

The message was too large to be sent automatically, as required by the socket options.

Effect

The call failed.

Recovery

Reduce the message size and retry the call.

ENAMETOOLONG (4131)

`ENAMETOOLONG`

Cause

The call specified a process or file name that exceeds the maximum allowable name length.

Effect

The call failed.

Recovery

Correct the process or file name and retry the call.

ENETDOWN (4116)

`ENETDOWN`

Cause

The network is down. The operation encountered a dead network.

Effect

The call failed.

Recovery

Contact the network manager.

ENETRESET (4118)

`ENETRESET`

Cause

The network dropped the connection because of a reset. The host you were connected to failed and rebooted.

Effect

The call failed, and all connections to the specified remote host were closed.

Recovery

Close the sockets using the `close` call. Reestablish the connections using the `socket`, `bind`, `connect`, and `accept` calls and retry the call.

ENETUNREACH (4117)

`ENETUNREACH`

Cause

The specified remote network was unreachable.

Effect

The interface is down.

Recovery

Retry the call.

ENOBUFS (4121)

`ENOBUFS`

Cause

There was not enough buffer space available to complete the call.

Effect

The call failed.

Recovery

Retry the call.

ENOMEM (4012)

`ENOMEM`

Cause

Insufficient memory was available to complete the call.

Effect

The call failed.

Recovery

Retry the call.

ENOPROTOOPT (4108)

`ENOPROTOOPT`

Cause

A call to `getsockopt`, `getsockopt_nw`, `setsockopt`, or `setsockopt_nw` specified an option that was unknown to the specified protocol.

Effect

The call failed.

Recovery

Specify the correct operation or protocol and retry the call.

ENOSPC (4028)

`ENOSPC`

Cause

The call required the addition of a filter and the adapter does not have sufficient memory to complete the request.

Effect

The call failed.

Recovery

Reduce the number of `connect` and/or `listen` calls.

ENOTCONN (4123)

`ENOTCONN`

Cause

The specified socket was not connected.

Effect

The call failed.

Recovery

Ensure that the socket is connected and retry the operation.

ENOTSOCK (4104)

`ENOTSOCK`

Cause

A socket operation was attempted on an object that is not a socket.

Effect

The call failed.

Recovery

Specify a valid socket and retry the operation.

ENXIO (4006)

`ENXIO`

Cause

The call specified an unknown device or the request was outside of the device capabilities.

Effect

The call failed.

Recovery

Correct the call using a known interface device or configure the desired interface device and retry the call.

EOPNOTSUPP (4111)

`EOPNOTSUPP`

Cause

The operation is not supported on a transport end point. For example, the application tried to accept a connection on a datagram transport end point.

Effect

The call failed.

Recovery

Specify a valid transport end point and retry the call.

EPERM (4001)

`EPERM`

Cause

The specified I/O control operation cannot be performed by a nonprivileged user. Only applications whose process access ID is in the SUPER group (user ID 255,n) can perform the operations that alter network parameters.

Effect

The call failed.

Recovery

Use the Subsystem Control Facility (SCF) ALTER command (or its programmatic equivalent), rather than socket calls. See the *TCP/IP Configuration and Management Manual* or the *TCP/IP Management Programming Manual* for a description of the ALTER command.

EPFNOSUPPORT (4112)

`EPFNOSUPPORT`

Cause

The specified protocol family is not supported. It has not been configured into the system or no implementation for it exists. The protocol family is used for the Internet protocols.

Effect

The call failed.

Recovery

Specify `AF_INET` and retry the operation.

EPIPE (4032)

`EPIPE`

Cause

A `write` or `send` call was attempted on a local socket that had been previously closed with the `shutdown` call.

Effect

The call failed.

Recovery

Reestablish the connection using the `socket`, `bind`, and `connect` calls and retry the `write` or `send` call.

EPROTONOSUPPORT (4109)

`EPROTONOSUPPORT`

Cause

The protocol specified in a call to `socket` or `socket_nw` is not supported.

Effect

The call failed.

Recovery

For protocol, specify a number in the range 0 to 255, excluding the values 1, 6, and 17 (the values assigned to ICMP, TCP, and UDP, respectively).

EPROTOTYPE (4107)

`EPROTOTYPE`

Cause

The protocol specified does not support the semantics of the socket type requested.

Effect

The call failed.

Recovery

Retry the call using the proper protocol type.

ERANGE (4034)

`ERANGE`

Cause

A numeric specification in the call is not within the allowable range.

Effect

The call failed.

Recovery

Correct the faulty specification and retry the call.

ESHUTDOWN (4124)

ESHUTDOWN

Cause

The operation could not be performed because the specified socket was already shut down.

Effect

The call failed.

Recovery

Reopen the remote socket using the `open`, `bind`, and `accept` calls. Reestablish the connection using a call to `connect` or `connect_nw`.

ESOCKTNOSUPPORT (4110)

ESOCKTNOSUPPORT

Cause

The socket type specified in a call to `socket` or `socket_nw` is not supported.

Effect

The call failed.

Recovery

Specify *`socket_type`* as `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`.

ESRCH (4003)

ESRCH

Cause

An `accept_nw2` call was issued on a socket that had been shut down or closed.

Effect

The call failed.

Recovery

Close all sockets associated with the connection. Attempt to reestablish the connection with the `socket_nw`, `bind_nw`, `accept_nw`, `socket_nw`, and `accept_nw2` calls. Each of these calls should be followed by an AWAITIOX call to ensure proper completion.

ETIMEDOUT (4126)

ETIMEDOUT

Cause

The connection timed out before the operation completed.

Effect

The call failed.

Recovery

Close the local socket. Rebuild the local socket using the `socket` and `bind` calls. Call `connect` or `connect_nw` to reestablish the connection.

EWOULDBLOCK (4101)

EWOULDBLOCK

Cause

A recv(MSG_OOB) or recv_nw(MSG_OOB) call was issued with the MSG_OOB flag set, but there was no out-of-band data to read.

Effect

The call failed.

Recovery

Execute a recv or recv_nw call without setting the MSG_OOB flag. If the recv or recv_nw call fails with an EHAVEOOB value in errno, call recv or recv_nw with the MSG_OOB flag set.

Table 21 Socket Errors by Number and Name

Error Number	Error Name	Error Number	Error Name
4001	EPERM	4109	EPROTONOSUPPORT
4003	ESRCH	4110	ESOCKTNOSUPPORT
4004	EINTR	4111	EOPNOTSUPP
4005	EIO	4112	EPFNOSUPPORT
4006	ENXIO	4113	EAFNOSUPPORT
4009	EBADF	4114	EADDRINUSE
4012	ENOMEM	4115	EADDRNOTAVAIL
4013	EACCES	4116	ENETDOWN
4014	EFAULT	4117	ENETUNREACH
4017	EEXIST	4118	ENETRESET
4022	EINVAL	4119	ECONNABORTED
4024	EMFILE	4120	ECONNRESET
4028	ENOSPC	4121	ENOBUFS
4032	EPIPE	4122	EISCONN
4034	ERANGE	4123	ENOTCONN
4101	EWOULDBLOCK	4124	ESHUTDOWN
4102	EINPROGRESS	4126	ETIMEDOUT
4103	EALREADY	4127	ECONNREFUSED
4104	ENOTSOCK	4128	EHOSTDOWN
4105	EDESTADDRREQ	4129	EHOSTUNREACH
4106	EMSGSIZE	4131	ENAMETOOLONG
4107	EPROTOTYPE	4195	EHAVEOOB
4108	ENOPROTOOPT	4196	EBADSYS

Index

D

E

F

G

H

I

L

M

N

O

U

W