System Software Library

Guardian Application Conversion Guide

Abstract This guide describes how to convert a TAL, COBOL85, or Pascal application or a TACL program to use the extended features of the Tandem NonStop Kernel. It is intended for application programmers.

Part Number 096047

Edition Second

Published September 1993

Product Version Guardian D20

Release ID D20.00

Supported Releases This manual supports D20.00 and all subsequent releases until otherwise indicated in a new edition.

Document History

Edition	Part Number	Product Version	Earliest Supported Release	Published
First	069808	Guardian 90 D10	N/A	January 1993
Second	096047	Guardian D20	D20.00	September 1993

New editions incorporate any updates issued since the previous edition.

A plus sign (+) after a release ID indicates that this manual describes function added to the base release, either by an interim product modification (IPM) or by a new product version on a .99 site update tape (SUT).

Export Statement Export of the information contained in this manual may require authorization from the U. S. Department of Commerce.

Examples Examples and sample programs are for illustration only and may not be suited for your particular purpose. Tandem does not warrant, guarantee, or make any representations regarding the use or the results of the use of any examples or sample programs in any documentation. You should verify the applicability of any example or sample program before placing the software into productive use.

Ordering Information For manual ordering information: Domestic U.S. customers, call 1-800-243-6886; international customers, contact your local sales representative.

New and Changed Information

The following changes have been made to the *Guardian Application Conversion Guide* for this edition:

- ☐ The name of this guide has been changed from the *Guardian 90 Operating System Application Conversion Guide* to the *Guardian Application Conversion Guide*.
- ☐ Section 8, "Converting Other Parts of an Application," has been updated to include information on converting applications to use the ADDRESS_DELIMIT_ and SEGMENT_USE_ procedures.
- ☐ Section 9, "Converting to TNS/R Systems," is new. Most of the information in this section applies to all TNS/R systems. Some of the information on data swap file size, however, applies only to D20 TNS/R systems.
- ☐ The list of system procedures in Appendix A has been updated to include the ADDRESS_DELIMIT_ and SEGMENT_USE_ procedures.
- ☐ Appendix D, "Considerations for Migrating Any Application," contains the following additions:
  - ☐ A discussion of condition codes.
  - ☐ A discussion of the MLAM PORT interface.
  - ☐ A discussion of bounds checking of reference parameters.
  - ☐ A discussion of ensuring adequate pool space.
- ☐ The glossary has been updated to add TNS/R terms that are used in this edition.

Contents

Section 2 Conversion Concepts (continued)

Section 3 Converting TAL Applications

Section 3 Converting TAL Applications (continued)

Section 4 Converting COBOL85 Applications

Section 5 Converting C Applications

Section 5 Converting C Applications (continued)

Section 6 Converting Pascal Applications

Section 6 Converting Pascal Applications (continued)

Section 7 Converting TACL Programs

Section 8 Converting Other Parts of an Application

Section 8 Converting Other Parts of an Application (continued)

Section 9 Converting to TNS/R Systems

Tables

About This Guide

This guide describes how to convert a TAL, COBOL85, C, or Pascal application or a Tandem Advanced Command Language (TACL) program to use the extended features of the D-series operating system.

Audience

This guide is intended for an application programmer who is converting an application. The reader should be familiar with:

- ☐ The language (TAL, COBOL85, C, Pascal, or TACL) used for the application
- ☐ The Guardian programmatic interface, if the application calls Guardian procedures or reads system messages from $RECEIVE
- ☐ The Tandem requester-server approach to application programming, if the application is a requester or a server

How to Use This Guide

This guide is designed to help application programmers who want to convert their programs to take advantage of the higher limits of the D-series operating system. It is published as part of the D20 release, but there is more than one way to migrate to D20.

- ☐ You might be migrating to D20 from a C-series release. In that case, this guide provides you with the information necessary to convert your applications as you migrate from a C-series system to a D-series system.
- ☐ You might be migrating to D20 from the D10 release. In that case, your applications already run successfully on a D-series system, and you might already have converted them to take advantage of some or all of the higher limits of the D-series operating system. This guide will assist you in converting any parts of your application that have not already been converted and that you want to convert at this time.

  There are only a few things to convert at D20 that are new since D10. See "New and Changed Information" at the beginning of this guide.
- ☐ You might also be migrating to D20 running on a TNS/R system. Because TNS/R systems were supported by C-series operating systems beginning at C30.06, you might be coming from a TNS/R system already; in that case, there should be few changes specific to TNS/R systems that you need to make. See "New and Changed Information" at the beginning of this guide. If you are migrating from a TNS system, however, you should read Section 9, "Converting to TNS/R Systems."

Organization

This guide covers these topics:

Section 1 explains why Tandem developed the D-series operating system and summarizes the differences between a C-series and D-series operating system that affect an application programmer. It also provides an approach to converting an application and directs the reader in how to find the required information in the rest of the guide.

Section 2 provides details of some concepts associated with conversion. It expands on the differences between C-series and D-series systems and provides a comparison of C-series syntax with D-series syntax.

Section 3 describes how to convert a TAL application. This section also applies to a COBOL85, C, or Pascal application that calls Guardian procedures.

Section 4 describes how to convert a COBOL85 application.

Section 5 describes how to convert a C application.

Section 6 describes how to convert a Pascal application.

Section 7 describes how to convert a TACL application.

Section 8 describes how to convert the parts of an application that are not described in Sections 3 through 7. This section applies to TAL, COBOL85, C, Pascal, and TACL applications.

Section 9 describes how to convert an application to run on TNS/R systems. This section applies mainly to TAL programs; one subsection applies to C programs as well.

Appendix A lists each C-series Guardian procedure and the corresponding D-series Guardian procedure, if one exists.

Appendix B lists the C-series and D-series user-level system messages that an application can read from $RECEIVE.

Appendix C describes compatibility between processes on C-series systems and D-series systems in a network.

Appendix D describes the changes that you might have to make to your application whether you want the D-series enhancements or not. For most applications, these changes are not necessary.

Related Manuals

The required and optional manuals are listed below and shown in Figure 1.

Required Manuals

The *Introduction to D-Series Systems* is an overview of features that are unique to the D-series operating system.

The D-series *Guardian Programmer's Guide* describes how to use the D-series Guardian programmatic interface.

The *Guardian Procedure Calls Reference Manual* describes syntax and programming considerations for C-series and D-series Guardian procedures.

The *Guardian Procedure Errors and Messages Manual* describes error codes, error lists, system messages, and trap numbers for C-series and D-series Guardian procedures.

Required Distributed Systems Management (DSM) Manuals

The *Subsystem Programmatic Interface (SPI) Programming Manual* describes SPI and how to use it in a DSM application.

The *Event Management Service (EMS) Manual* describes EMS, which allows an application to collect, process, distribute, and generate event messages.

- ☐ The *Tandem NonStop Kernel Event Management Programming Manual* describes *Tandem NonStop Kernel* event messages.

Required Language Reference Manuals

- ☐ *Transaction Application Language (TAL) Reference Manual*
- ☐ *COBOL85 Reference Manual*
- ☐ *C Reference Manual*
- ☐ *Pascal Reference Manual*
- ☐ *TACL Reference Manual*

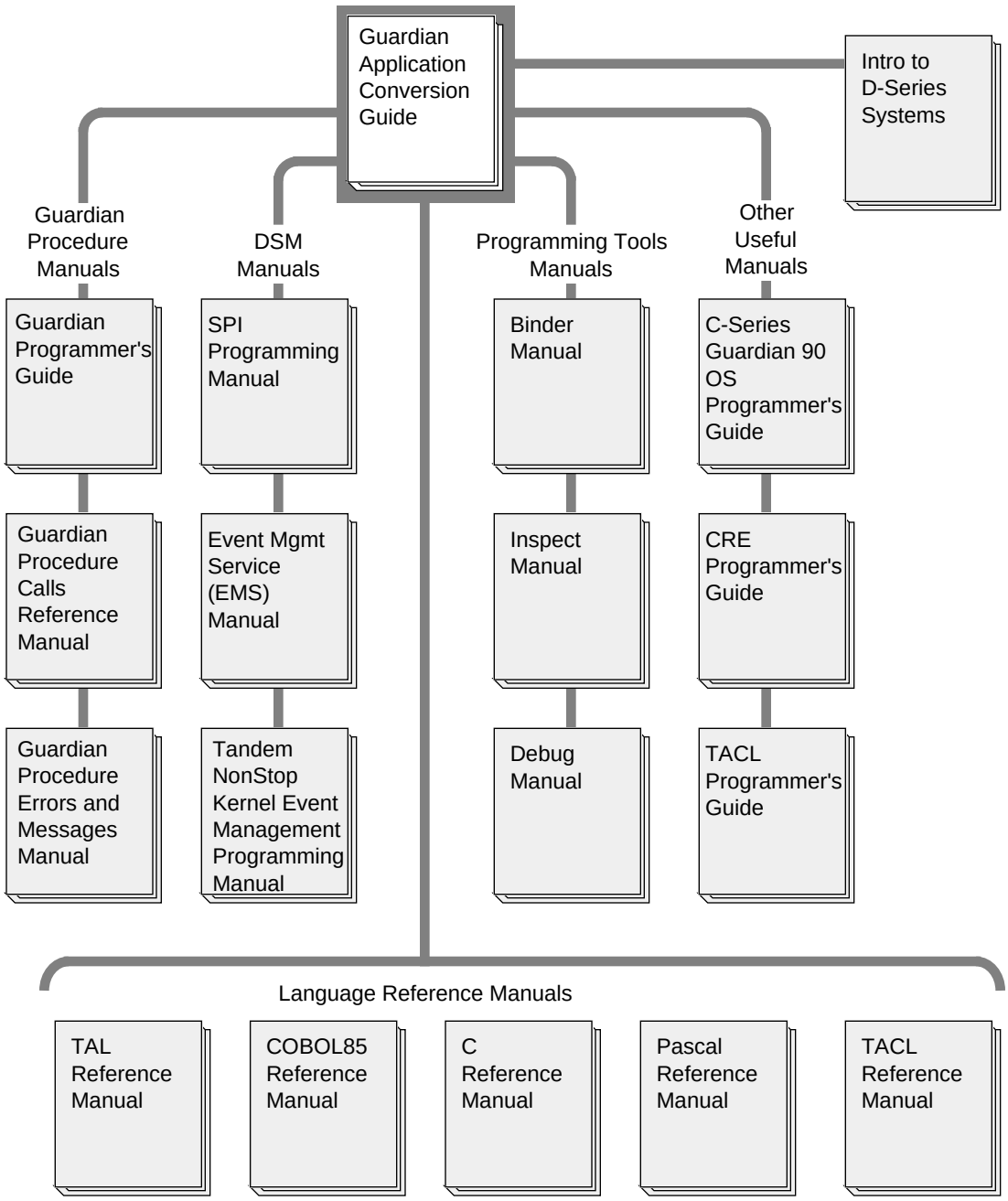
Required Program Development Manuals

The *Binder Manual* describes Binder, an interactive linker that allows you to examine, modify, and combine object files and to generate load maps and cross-reference listings.

The *Inspect Manual* describes the Inspect program, which is both a source-level and a machine-level interactive debugger.

The *Debug Manual* describes Debug, a machine-level interactive debugger.

Figure 1. Related Manuals



Optional Manuals

The C-series *Guardian Programmer's Guide* describes how to use the C-series Guardian programmatic interface.

The *TAL Programmer's Guide* provides TAL programming information for the less experienced programmer who will need to read this guide before using the *Transaction Application Language (TAL) Reference Manual*.

The *Common Run-Time Environment (CRE) Programmer's Guide* describes the Common Run-Time Environment for TAL, COBOL85, C, and Pascal applications.

The *TACL Programmer's Guide* describes how to write TACL programs.

Notation Conventions

General Syntax Notation

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

UPPERCASE LETTERS

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

```
MAXATTACH
```

lowercase italic letters

Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

```
file-name
```

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 may 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:

```
LIGHTS [ ON             ]
       [ OFF            ]
       [ SMOOTH [ num ] ]

K [ X | D ] address-1
```

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 may 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-1 [ , 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 entered 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 enter 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 the following example, there are no spaces 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 / ] CONTROLLER

   [ , 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
```

1 Introduction

Most TAL, COBOL85, C, or Pascal applications or Tandem Advanced Command Language (TACL) programs written for the C-series operating system can run under the D-series operating system without modification. However, to take advantage of the higher limits of the D-series operating system, an application might need to be converted as described in this manual.

Also, most applications written to run on TNS systems can run on TNS/R systems without modification. However, modifications might be required in some programs, particularly in privileged TAL programs. This manual also provides information about converting applications to run on TNS/R systems.

This section introduces the D-series operating system and summarizes the differences between the C-series and the D-series operating systems that affect an application program. This section also provides an approach to converting an application and discusses conversion options.

Why a New Operating System?

Tandem designed the C-series operating system for earlier computer systems. Newer systems such as the Cyclone system are faster and more sophisticated. Therefore, when running a C-series operating system, newer computer systems sometimes encounter these limitations:

- ☐ CPUs are not used efficiently.

  Newer Tandem systems execute instructions faster than older systems. When running a C-series operating system, which supports only 256 concurrent processes per CPU, a CPU in a newer system can become I/O bound (that is, all processes are waiting for an I/O operation to finish). Sometimes the CPU becomes idle and must wait before it can continue executing instructions.

- ☐ I/O configurations are limited.

  While earlier systems have only one I/O channel per CPU, newer systems allow a maximum of four channels per CPU. Usually, each I/O device requires an I/O process (IOP), which is a system process that controls the device. Increasing the number of I/O devices in a system also requires an increase in the number of IOPs that are running concurrently in a CPU. Thus, when running a C-series operating system, which allows only 256 concurrent processes per CPU, a newer system might not be able to support its maximum number of I/O devices.

The D-series operating system increases the number of concurrent processes per CPU from 256 to an architectural limit of 65,534. The actual limit is at most 2500 and might be less, depending on availability of segments, time-list elements (TLEs), and memory. Increasing the number of concurrent processes per CPU:

- ☐ Improves the efficiency of each CPU
- ☐ Allows for more IOPs so that larger I/O configurations can be supported

The D-series operating system also supports larger I/O configurations by increasing the software limits for the maximum number of:

- ☐ I/O devices and named processes from 4,096 to 65,535 per system
- ☐ I/O subdevices per device from 256 to a theoretical limit of 65,535 depending on the subsystem
- ☐ Opens per subdevice from 16 to a theoretical limit of 65,535, again depending on the subsystem
- ☐ Opens per disk volume from 4,096 to 32,767

Again, these values are the architectural limits. The actual limits depend on the available resources of the system.

What Are the Differences?

The major differences between the C-series operating system and the D-series operating system that affect an application program are:

- ☐ More concurrent processes per CPU

  The D-series operating system allows more than 256 concurrent processes per CPU; a C-series operating system allows a maximum of 256 processes per CPU.

- ☐ New system procedures

  New user-callable system procedures are available for converted applications in order to support the extended system limits. These procedures use new naming and error-return conventions.

- ☐ New file-name format

  The D-series file-name format for disk file names, device names, and process file names is a variable-length string. The string length is specified as a separate integer value.

- ☐ New process identifiers

  The D-series process file name is a variable-length string that replaces the C-series process file name. The D-series process handle is a 20-byte structure that replaces the C-series 4-word process ID (or CRTPID).

- ☐ New system messages

  New user-level system messages are available for converted applications to read from $RECEIVE.

- ☐ Improved performance for I/O operations

  The D-series operating system can improve the performance of I/O operations by directly transferring data between an application's I/O buffers and the I/O device without having to allocate an intermediate buffer in the application's process file segment (PFS).

- ☐ New Distributed Systems Management (DSM) tokens

  The Event Management Service (EMS) and the Subsystem Programmatic Interface (SPI) have new tokens and filter functions for DSM applications.

- Other changes

  To support its extended features, the D-series operating system also provides new file-system error numbers, file-system error lists, and object-file attributes.

Section 2, "Conversion Concepts," describes these differences in detail.

Note There are two programmable interfaces to a D-series system:

- The C-series-compatible interface. You cannot make use of the extended system limits using this interface. Entities that describe parts of the C-series-compatible interface are generally referred to in this guide as C-series entities.
- The D-series enhanced interface, which supports the new system limits as described earlier. Entities that form parts of the D-series enhanced interface are referred to in this guide as D-series entities.

A program that runs on a D-series system can use either or both of these interfaces.

More Concurrent Processes per CPU

Although the D-series operating system increases several software limits, the limit that most directly affects an application is the number of concurrent processes per CPU. A D-series operating system can support an architectural limit of 65,534 concurrent processes per CPU. The actual number of concurrent processes per CPU depends on the system's available resources, such as virtual memory.

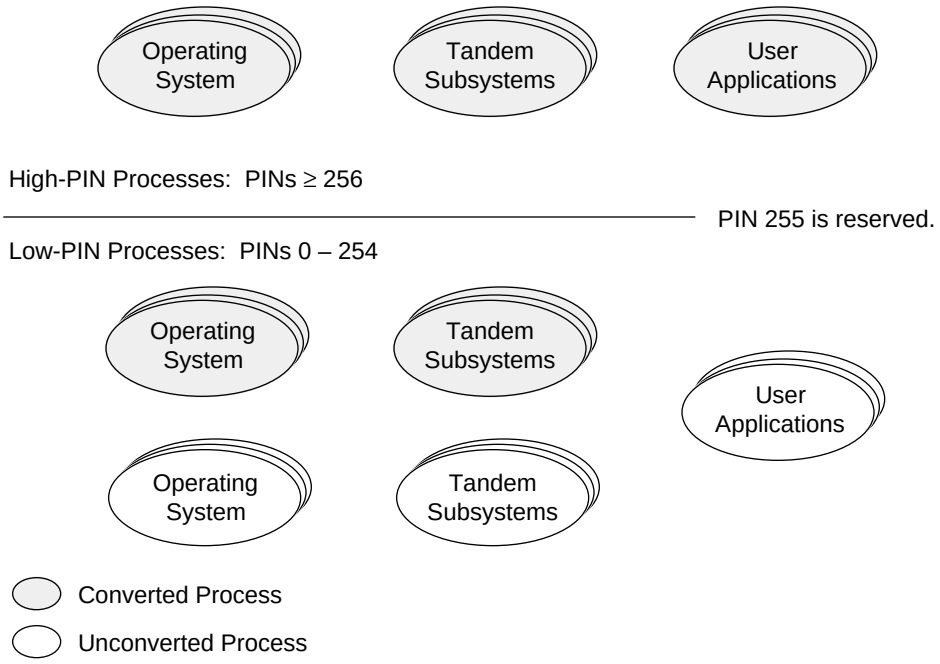
The system identifies a process by a process identification number (PIN). When the system creates a new process, it assigns a PIN to the process. A C-series PIN ranges from 0 through 255 (and therefore can be represented in an 8-bit field). A D-series PIN ranges from 0 through the maximum number of processes per CPU supported by the system. A D-series PIN falls in these categories:

- A low PIN ranges from 0 through 254.
- A high PIN ranges from 256 through the maximum number supported for the CPU.

PIN 255 is never assigned to a process. It is sometimes used in a synthetic process ID, which is described in Appendix C.

Figure 1-1 shows various converted and unconverted processes running in a CPU in the D-series operating system environment.

Figure 1-1. D-Series Operating System Environment



Conversion Strategy

Converting application programs includes those activities that allow you to change C-series applications to take advantage of the D-series enhancements. *Migrating* application programs includes those activities that allow you to run C-series applications on D-series systems.

Almost all applications designed to run under the C-series operating system can migrate without modification to run under the D-series operating system by continuing to use the C-series-compatible interface. The exceptions are few and are detailed in Appendix D, "Considerations for Migrating Any Application." Therefore, there is usually no need to do any conversion unless your system has reached or is likely to reach the C-series limit of 255 concurrent processes.

If you might reach the C-series limit of 255 processes, then you need to move some processes into high PINs. Tandem recommends converting your applications if, after placing Tandem and third-party products in high PINs, you still need more process control blocks (PCBs) in the low-PIN range than are available.

To move processes into a high PIN requires conversion, because the process identifiers used in the C-series-compatible interface allow for only 8-bit PINs.

The amount of conversion you need to do depends on the type of application. Single-process applications, for example, are easier to convert than multiple-process applications, and some multiple-process applications need more conversion effort than others.

Converting Single-Process Applications

Interprocess communication and process management might need a moderate to large amount of conversion effort. Single-process applications need the least effort to convert because they communicate with few, if any, other processes. They also do not keep track of what processes open them. They do not create other processes because if they did, they would not be single-process applications.

Word processors and calculators are typical examples of single-process applications. They are also applications that are executed by a large number of users. Since each execution of these processes requires a separate PIN a lot of low PINs can be freed by converting such processes to run at high PINs.

Converting Multiple-Process Applications

The amount of effort needed to convert multiple-process applications ranges from small to extensive. As the amount of interprocess communication and management increases, so does the conversion effort needed. The following list of applications that contain multiple processes appears in order of the amount of effort, from low to high, it would take to convert these applications:

- ☐ COBOL85 program with no ENTER TAL statements
- ☐ Most requester processes
- ☐ Server processes that do not track processes that open them
- ☐ Server processes that care about their creator's identity
- ☐ Server processes that do track processes that open them
- ☐ Monitor process pairs that create and track other processes

The following paragraphs outline the conversion effort required for each of these application types.

COBOL85 Program With No ENTER TAL Statements

If a COBOL85 program contains no ENTER TAL statements, you do not have to convert C-series procedure calls to D-series procedure calls. You do need to use the Common Run-Time Environment (CRE), however, to run a COBOL85 program at a high PIN.

Most Requester Processes

A requester process opens a server, makes a request, and gets a reply. If the server is named, then the requester does not have to understand the server's CPU and PIN; the system takes care of the interprocess communication.

If the server is not named, some changes are necessary. It is probably easier to run the server named than to change the requester.

Server Processes That Do Not Track Processes That Open Them

A server process that does not read system messages from its $RECEIVE file probably does not need to process the CPU and PIN of the requester, because it does not keep a table of processes that open it.

Server Processes That Care About Their Creator's Identity

Some processes check the identity of the process that creates them. Such a process might, for example, compare that process identifier with the sender of the Startup message to make sure that the sender is also the creator.

Some changes might be necessary if the creator runs at a high PIN; the server needs to be able to obtain the identity of a high-PIN creator and a high-PIN opener.

Server Processes That Do Track Processes That Open Them

This type of server keeps a table of processes that open it. If the server uses the CPU and PIN to uniquely identify processes that open it, the server must be converted to understand the PIN of a high-PIN process. In many cases, however, all that is necessary is to set the HIGHREQUESTERS attribute for the server.

Monitor Process Pairs That Create and Track Other Processes

Monitor process pairs create, monitor, and control other processes. They also keep tables of information about the attributes of these processes. If they create and track high-PIN processes, some changes to your application are required.

An Approach to Converting an Application

A typical application consists of several processes that communicate with each other to achieve a specific objective. Converting an application involves changing parts of each process. If you are converting an application, you might consider the approach for the application shown below. This application consists of these processes:

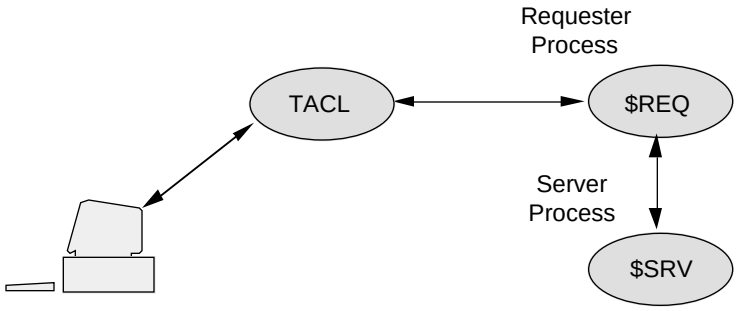
$REQ A requester process that creates, opens, and sends requests to the server process $SRV. $REQ also monitors and manages $SRV. Process $REQ is started from a terminal using the TACL RUN command. By default, TACL starts a new process at a high PIN if possible.

$SRV A server process that is opened by the requester process $REQ and receives requests from $REQ. $SRV also maintains an opener table to track its openers.

The goal of the approach listed below and shown in Figure 1-2 is for both $REQ and $SRV to run at high PINs.

1. Convert the requester $REQ and the server $SRV to run at high PINs.
2. Convert the requester $REQ to create the server $SRV at a high PIN and then to manage the high-PIN $SRV process (for example, to suspend or stop it).
3. Convert the requester $REQ to open and send requests to $SRV and to monitor the high-PIN $SRV process.
4. Convert the server $SRV to receive requests from the requester $REQ and to maintain an opener table.
5. Convert other parts of the requester $REQ and server $SRV as needed, such as managing extended memory and files.

Figure 1-2. Sample Application to Be Converted




```
10> RUN reqfile /NAME $REQ,.../
```

Sections 3 through 7 describe the conversion of TAL, COBOL85, C, Pascal, and TACL applications, respectively. Section 8, "Converting Other Parts of an Application," describes converting parts of an application that can apply to any of these languages.

Conversion Options

The approach to conversion described in this guide provides you with enough information to convert everything in your application to use the D-series enhanced interface. Of course, not every application needs to have every part converted. Depending on your application, several options are available. The following list outlines some of the major enhancements you might need to make, starting with the easiest options and working to the most complex:

- ☐ Running your process at a high PIN
- ☐ Allowing your process to be created by a high-PIN process
- ☐ Opening a high-PIN process
- ☐ Allowing your process to be opened by a high-PIN process
- ☐ Creating and managing a high-PIN process

The following paragraphs indicate where in this guide you need to turn to find procedural information on the above options.

Running a Process at a High PIN

The following table identifies the parts of this guide that tell you how to convert an application to be capable of running at a high PIN.

For programs written in...	See the subsection entitled...	In Section...
TAL	Converting a TAL Program to Run at a High PIN	3
COBOL85	Converting a COBOL85 Program to Run at a High PIN	4
C	Converting a C Program to Run at a High PIN	5
Pascal	Converting a Pascal Program to Run at a High PIN	6

For information about using TACL to run processes at a high PIN, see Section 7, "Converting TACL Programs." These programs can be written in any language.

Allowing a High-PIN Creator

The following table identifies the parts of this guide that tell you how convert an application to be capable of being created by a high-PIN process.

For programs written in...	See Section ...	Subsection entitled...	If the process reads system messages, then, in the same section, see also...
TAL	3	Allowing a High-PIN Creator	
COBOL85	4	Setting the HIGHREQUESTERS Object-File Attribute	Converting a Server
C	5	Setting the HIGHREQUESTERS Object-File Attribute	Monitoring a High-PIN Server and Converting a Server
Pascal	6	Setting the HIGHREQUESTERS Object-File Attribute	Monitoring a High-PIN Server and Converting a Server

This operation does not apply to TACL programs.

Opening a High-PIN Process

To open a high-PIN process, see "Opening and Communicating With a High-PIN Server" in the section that corresponds to your programming language (Sections 3 through 6). This operation does not apply to TACL programs.

Allowing a High-PIN Opener

To allow a high-PIN opener, see "Being Opened by and Communicating With a High-PIN Requester" in the section that corresponds to your programming language (Sections 3 through 6). This operation does not apply to TACL programs.

Creating and Managing a High-PIN Process

The following tables identify the subsections of this guide that you should refer to for help in creating and managing high-PIN processes. Separate tables are included for TAL, COBOL85, C, and Pascal. These operations do not apply to TACL programs.

For TAL Programs

Step...	You might need to convert your application to...	As described in Section...	Subsection entitled...
1	Create any processes that you want at a high PIN using PROCESS_CREATE_	3	Creating a High-PIN Process
2	Create any processes that you want at a low PIN using PROCESS_CREATE_	3	Creating a Low-PIN Process
3	Create a backup process (if required) using PROCESS_CREATE_	3	Creating a High-PIN Process
4	Manage processes using the D-series enhanced procedures	3	Managing a High-PIN Process
5	Get information about processes using the PROCESS_GETINFO_ procedure	3	Getting Information About a High-PIN Process
6	Set process attributes using the PROCESS_SETINFO_ procedure	3	Setting Process Attributes for a High-PIN Process
7	Monitor the process status	3	Monitoring a High-PIN Server

For COBOL85 Programs

Step...	You might need to convert your application to...	As described in Section...	Subsection entitled...
1	Create any processes that you want at a high PIN using PROCESS_CREATE_	4	Creating a High-PIN Process
2	Create any processes that you want at a low PIN using PROCESS_CREATE_	4	Creating a High-PIN Process
3	Create a backup process (if required) using PROCESS_CREATE_ as for any high-PIN process	4	Creating a High-PIN Process
4	Manage processes using the D-series enhanced procedures	4	Managing a High-PIN Process
5	Get information about processes using the PROCESS_GETINFO_ procedure	3	Getting Information About a High-PIN Process
		and 4	Converting Guardian System Procedure Calls
6	Set process attributes using the PROCESS_SETINFO_ procedure	3	Setting Process Attributes for a High-PIN Process
		and 4	Converting Guardian System Procedure Calls
7	Monitor the process status	4	Converting the RECEIVE-CONTROL Paragraph

For C Programs

Step...	You might need to convert your application to...	As described in Section...	Subsection entitled...
1	Create any processes that you want at a high PIN using PROCESS_CREATE_	5	Creating a High-PIN Process
2	Create any processes that you want at a low PIN using PROCESS_CREATE_	5	Creating a High-PIN Process
3	Create a backup process (if required) using PROCESS_CREATE_	5	Creating a High-PIN Process
4	Manage processes using the D-series enhanced procedures	3	Managing a High-PIN Process
		and 5	Converting Guardian System Procedure Calls
5	Get information about processes using the PROCESS_GETINFO_ procedure	3	Getting Information About a High-PIN Process
		and 5	Converting Guardian System Procedure Calls
6	Set process attributes using the PROCESS_SETINFO_ procedure	3	Setting Process Attributes for a High-PIN Process
		and 5	Converting Guardian System Procedure Calls
7	Monitor the process status	5	Monitoring a High-PIN server

For Pascal Programs

Step...	You might need to convert your application to...	As described in Section...	Subsection entitled...
1	Create any processes that you want at a high PIN using PROCESS_CREATE_	6	Creating a High-PIN Process
2	Create any processes that you want at a low PIN using PROCESS_CREATE_	6	Creating a High-PIN Process
3	Create a backup process (if required) using PROCESS_CREATE_	6	Creating a High-PIN Process
4	Manage processes using the D-series enhanced procedures	3	Managing a High-PIN Process
		and 6	Converting Guardian System Procedure Calls
5	Get information about processes using the PROCESS_GETINFO_ procedure	3	Getting Information About a High-PIN Process
		and 6	Converting Guardian System Procedure Calls
6	Set process attributes using the PROCESS_SETINFO_ procedure	3	Setting Process Attributes for a High-PIN Process
		and 6	Converting Guardian System Procedure Calls
7	Monitor the process status	6	Monitoring a High-PIN Server

Converting Other Parts of an Application

If you convert your application programs to use the D-series enhancements, you might have to convert other parts of the application:

- ☐ Terminal I/O operations; specifically those that have an interface with TACL or process the BREAK key
- ☐ Sequential I/O operations
- ☐ Direct transfers during I/O operations
- ☐ Memory management for extended data segments, if your process shares an extended data segment with one or more processes

There are two additional memory-management procedure calls that you can convert your application to use: SEGMENT_USE_ and ADDRESS_DELIMIT_; however, conversion is optional. There are also some new procedure calls that make managing disk files easier, but converting disk file management applications is optional. For details on how to convert these other parts of an application, see Section 8, "Converting Other Parts of an Application."

Converting to Run on TNS/R Systems

Most TNS programs written for the C30 and D-series versions of the operating system can run on a TNS/R system without modification. Variances between TNS and TNS/R systems, however, might require modification in some programs, particularly in privileged TAL programs. Some C programs might require modification as well.

Variances between TNS and TNS/R systems exist in the following areas:

- ☐ Extended segment limit checking
- ☐ Overflow results
- ☐ Trap handlers that use the register stack
- ☐ Trap handlers that modify the P or E registers
- ☐ Privileged instructions
- ☐ Nonprivileged references to system global data
- ☐ Stack wrapping
- ☐ Odd-byte references
- ☐ Data swap file size

For details on these variances and on how to convert your application to run on a TNS/R system, see Section 9, "Converting to TNS/R Systems."

2 Conversion Concepts

This section describes the major concepts involved in converting an application to make use of the D-series enhancements. Specifically, this section describes:

- ☐ The use of the new Guardian procedures
- ☐ The new file-name format
- ☐ New process identifiers
- ☐ New system messages
- ☐ Performance improvements for I/O operations
- ☐ New Distributed Systems Management (DSM) tokens
- ☐ New file-system error numbers and error lists
- ☐ New object-file attributes
- ☐ Conversion considerations related to the Common Run-time Environment (CRE)

New Guardian Procedures

The D-series operating Guardian has many new user-accessible Guardian procedures. However, to ensure compatibility with unconverted applications, Tandem provides these new procedures in addition to the existing C-series procedures.

The D-series operating system does not include any new sequential I/O (SIO) procedures. The SET^FILE, CHECK^FILE, and OPEN^FILE procedures (and their LITERAL and DEFINE declarations) have been modified, but other SIO procedures are unchanged.

Tandem provides declarations for many of the options and structures used as parameters to the new Guardian procedures. Tandem uses the ZSYSDDL file to generate the ZSYSTAL, ZSYSCOB, ZSYSC, ZSYSPAS , and ZSYSTACL files for TAL, COBOL85, C, Pascal, and TACL applications, respectively. There is no equivalent file for TACL programs. To use the ZSYSDDL declarations, include the appropriate file (or sections of the file) in your source code.

This subsection describes in general the characteristics of the new procedures. For information about each procedure, refer to the *Guardian Procedure Calls Reference Manual*.

Note This guide uses the term **D-series Guardian procedure** to indicate a Guardian procedure that is new in the D-series release. The term **C-series Guardian procedure** is used to refer to a C-series-compatible Guardian procedure that can be called on either a D-series system or a C-series system.

Naming Conventions

A D-series procedure name has an underscore (_) after each part, including the last part. This convention allows you to give your own procedures names that do not conflict with Tandem procedure names provided that you do not end your procedure names with an underscore. A D-series procedure name follows one of these format conventions:

```
[module_]object_action_[CHKPT_]
```

*module_*

is the module or subsystem that owns the object. Examples are MBCS_ and SPI_.

*object_*

is the object that the procedure acts upon. Examples are FILE_ , PROCESS_ , and SEGMENT_.

*action_*

is the action that the procedure takes on the object. Examples are CREATE_ , STOP_ , and GETINFO_.

CHKPT_

specifies that the CHECKMONITOR procedure executes the procedure for the backup process of a process pair. For example, a primary process calls FILE_OPEN_CHKPT_ to open a file for its backup process.

```
type-1_TO_type-2_
```

*type-1*_TO_*type-2_*

specifies the conversion from *type-1_* to *type-2_*. Examples are CRTPID_TO_PROCESSHANDLE_ and PROCESSHANDLE_TO_FILENAME_.

Parameter Conventions

The D-series Guardian procedures use new input and output parameter conventions, including:

- ☐ Variable-length string parameters for file names, node names, and process descriptors
- ☐ 20-byte process handles that you use to identify processes to process-control procedures such as PROCESS_ACTIVATE_ and PROCESS_STOP_

The following paragraphs describe these new conventions.

String Parameters

The D-series procedures use a variable-length string for:

- ☐ All kinds of file names, including disk file names, device file names, and D-series process file names. A D-series process file name parameter represents either a named or unnamed process or a named process pair.
- ☐ Node names.
- ☐ Process-descriptor parameters, which are a specific form of D-series process file name returned by Guardian procedures.

Declaring String Parameters. Declare a string parameter as a reference parameter with its length specified as a separate integer value. The ZSYSTAL, ZSYSCOB, ZSYSC, and ZSYSPAS files contain declarations that you can use to declare string parameters. In a TAL program, you might declare a file name, a node name, and a process descriptor as follows:

```
!  File name declaration

STRING .file^name[0:ZSYS^VAL^LEN^FILENAME-1];
INT    file^name^length;

!  Node name declaration

STRING .node^name[0:ZSYS^VAL^LEN^SYSTEMNAME-1];
INT    node^name^length;

!  Process descriptor declaration

STRING .descriptor[0:ZSYS^VAL^LEN^PROCESSDESCR-1];
INT    descriptor^length;
```

A file-name input parameter must fill the entire string for the number of characters specified by the length and must not include any leading or trailing blanks or null characters (that is, the name buffer can include blanks or null characters only after the specified length).

The letters in a file name can be uppercase or lowercase (or a mixture of both cases); the Guardian procedures are not case-sensitive with regard to the letters.

For other examples of TAL, COBOL85, C, and Pascal file-name and process-descriptor string declarations, refer to Sections 3 through 6.

Specifying the String Parameter Length. You specify the length of a string parameter depending on whether the string is being used as an input-only, output-only, or input/output parameter as described below and as shown in the hypothetical procedure calls:

Input only — The string parameter is followed by a colon (:) and an integer input parameter that specifies the actual length in bytes of the string. Passing the string length as zero is equivalent to omitting the string parameter. For example:

```
error := PROC_CALL_ (file^name:file^name^length);    ! i:i
```

Output only — The string parameter is followed by a colon (:) and an integer input parameter that specifies the maximum length in bytes of your return buffer. Passing the return buffer length as zero is equivalent to omitting the string parameter.

The system returns the actual length of the string in a separate integer output parameter. For example:

```
error := PROC_CALL_ (file^name:max^buffer^length,    ! o:i
                     file^name^length);              ! o
```

Input/output — The string parameter is followed by a colon (:) and an integer input parameter that specifies the maximum length in bytes of your return buffer. Passing the return buffer length as zero is equivalent to omitting the string parameter.

A separate integer parameter contains the current string length. You set this parameter to the input parameter string length in bytes, and the system returns the length of the output parameter string. For example:

```
error := PROC_CALL_ (file^name:max^buffer^length,    ! i,o:i
                     file^name^length);              ! i,o
```

Process-Handle Parameters

In process-control procedures such as PROCESS_ACTIVATE_ or PROCESS_STOP_ , a 20-byte process handle identifies the target process. Declare a process handle as a reference parameter. The ZSYSTAL, ZSYSCOB, ZSYSC, and ZSYSPAS files contain declarations that you can use to declare process-handle parameters. An example of a TAL declaration is:

```
INT .process^handle[0:ZSYS^VAL^PHANDLE^WLEN-1];
```

Null Process Handle. Some procedures accept or return a null process handle, which has a -1 in each word. For example, a process can obtain information about itself by calling the PROCESS_GETINFO_ procedure with a null process handle input parameter. The PROCESS_GETPAIRINFO_ procedure returns a null process handle output parameter for the backup process if one does not exist.

To set a process handle parameter to a null value, set each word to -1. An example of a TAL null process handle is:

```
process^handle ':=' [ZSYS^VAL^PHANDLE^WLEN * [-1]];
```

However, when you check a process handle output parameter, you need check only the first word for a -1. If the first word of a process handle output parameter is -1, the parameter contains a -1 in each word and is a null process handle.

Error-Return Conventions

The D-series procedures do not use the condition code (CC) setting to indicate an error. Each procedure returns an integer error or status value. If an error condition contains more information than the procedure can return in an integer parameter, the procedure returns additional information in an integer *error-detail* parameter.

This hypothetical TAL example shows the *error* value and the *error-detail* parameter:

```
...
INT error;
INT error^detail;

...

error := PROC_CALL_ (parameter-1,
                     parameter-2,
                     parameter-3,
                     error^detail);
...
```

Depending on the procedure you call, the *error* value contains either a file-system error number or one of the following values:

0	The procedure was successful.
1	A file-system error occurred; *error-detail* contains the file-system error number.
2	A parameter error occurred. For example, a required parameter is missing.
3	A bounds error occurred on a reference parameter.
4 through *n*	Another error occurred; in some cases *error-detail* contains additional information. The value and interpretation of *n* depend on the procedure you are calling.

To avoid problems in future releases, your application should treat *error* values other than the ones currently defined for each procedure as undefined rather than invalid. Always include an OTHERWISE (or equivalent) clause in your application when checking error values.

If a procedure has more than one input parameter, a parameter error (*error* = 2) or a bounds error on a reference parameter (*error* = 3) can occur on more than one parameter. In this case, the *error-detail* parameter contains the ordinal number of

the first parameter that the system detected as causing an error (provided the procedure returns the *error-detail* parameter). If more than one parameter causes an error, *error-detail* does not necessarily point to the lowest-numbered parameter.

When determining the ordinal number, the string parameter and its length (for example, *file^name:file^name^length*) are treated as a single parameter.

New File-Name Format

A Tandem file name identifies a permanent or temporary disk file, an I/O device, or a process. A D-series file name is a variable-length string with its length in bytes specified as a separate integer value. For compatibility with unconverted applications, the D-series operating system also uses the 12-word internal-format file name used on C-series systems.

D-series disk file names and device names are described in this subsection. The D-series process file name, which supersedes the C-series process file name, is described under "New Process Identifiers" later in this section.

Disk File Names

D-series disk file names identify permanent or temporary disk files. A D-series disk file name uses the same format as a C-series external disk file name except as described below. Examples of valid D-series disk file names are:

```
REPORT                      ! Single file ID
FY90.MEMBERS                ! Subvolume and file ID
$USERS.PAYROLL.LEVEL2       ! Volume, subvolume, and file ID
\LONDON.$DISK4.ACCTS.JAN89  ! Fully qualified file name
$DISKVOL.#1234567           ! Temporary file name
```

Advantages of D-Series Disk File Names

D-series disk file names have the following advantages over C-series disk file names:

- The D-series Guardian procedures accept and return file-name string parameters rather than the C-series 12-word internal-format file-name parameters. You are not required to convert a D-series file name from external to internal format before you call a D-series Guardian procedure.
- The D-series Guardian procedures automatically expand a partially qualified D-series file name to a fully qualified file name, including the node (system) name, from the =_DEFAULTS DEFINE VOLUME attribute. You are not required to call the FNAMEEXPAND procedure to expand the name before you call a procedure.
- A D-series file name is suitable to display or print without any conversion. You are not required to call the FNAMECOLLAPSE procedure to convert the name to a suitable format before you display or print it.
- A remote D-series disk file name can have eight-character volume or device names (seven letters or digits after the dollar sign). A remote C-series disk-file volume or device name is restricted to seven characters including the dollar sign.

C-Series and D-Series Disk File Name Differences

A D-series disk file name is similar to an external-format disk file name used on C-series systems except for the following differences:

- Subvolume defaulting is not allowed. A D-series permanent disk file name does not allow subvolume defaulting. For example, this disk file name is invalid on D-series systems:

```
$DISKVOL.MYFILE            ! Volume name and file ID
```

- Temporary file names are longer. A D-series temporary file name can have up to seven digits after the pound sign (#). A C-series temporary file name has only four digits after the pound sign.
- Remote volume names can be longer. A converted process on a D-series system can identify remote disk files with eight-character volume names (seven characters after the dollar sign) on other D-series systems in a network. A process on a C-series system can identify a remote disk file with a maximum of seven characters in the volume name (six characters after the dollar sign) on other C-series systems in a network.

  However, a converted process on a D-series system cannot identify a remote C-series file with an eight-character volume name. For more information about the compatibility of C-series and D-series disk file names, refer to Appendix C, “System Compatibility.”

Partially Qualified File Names

If you call a D-series Guardian procedure, the D-series operating system expands a partially qualified file name using the current settings, including the node name, from the =_DEFAULTS DEFINE VOLUME attribute.

For example, suppose the =_DEFAULTS DEFINE VOLUME attribute is:

```
\SYSTEM.$VOL1.SUBVOL1
```

The D-series operating system expands a partially qualified file name as shown below. The parts of each expanded file name that are taken from the =_DEFAULTS DEFINE are shown in uppercase letters.

Partially Qualified File Name	File Name After Expansion
`fileid`	`\SYSTEM.$VOL1.SUBVOL1.fileid`
`subvol2.fileid`	`\SYSTEM.$VOL1.subvol2.fileid`
`$vol2.subvol2.fileid`	`\SYSTEM.$vol2.subvol2.fileid`
`\sys2.subvol2.fileid`	`\sys2.$VOL1.subvol2.fileid`
`\sys2.fileid`	`\sys2.$VOL1.SUBVOL1.fileid`

Table 2-1 shows a comparison of the programmatic representation of C-series and D-series disk file names.

Table 2-1. C-Series and D-Series Programmatic Representation of Disk File Names

Form	C-Series Programmatic Representation	D-Series Programmatic Representation
Permanent disk file (without a specified node name)[1]	*file-name* [0:3] = Volume name *file-name* [4:7] = Subvolume name *file-name* [8:11] = File ID	[[*volume*.]*subvolume*.]*file-id*
Network permanent disk file	*file-name* [0].<0:7> = ASCII "\" *file-name* [0].<8:15> = System number *file-name* [1:3] = Volume name [2] *file-name* [4:7] = Subvolume name *file-name* [8:11] = File ID	[*node-name*.][[*volume*.]*subvolume*.]*file-id* [3]
Temporary disk file (without a specified node name)[1]	*file-name* [0:3] = Volume name *file-name* [4:11] = Temporary file ID	[*node-name*.][*volume*.]#*temporary-file-number*
Network temporary disk file	*file-name* [0].<0:7> = ASCII "\" *file-name* [0].<8:15> = System number *file-name* [1:3] = Volume name [2] *file-name* [4:11] = Temporary file ID	[*node-name*.][*volume*.]#*temporary-file-number* [3]

1 The D-series operating system expands a partially qualified D-series file name using the current settings, including the node name, from the =_DEFAULTS DEFINE VOLUME attribute. A file name that does not include a node name is therefore not necessarily a local file name on a D-series system.

2 On C-series systems, a process can identify a remote volume name with a maximum of six characters. A remote volume name on a C-series system does not contain a dollar sign in the programmatic representation.

3 On D-series systems, a converted process can identify a remote volume name with two to eight characters including the dollar sign on other D-series systems but not on C-series systems.

Device Names

A device name identifies an I/O device such as a terminal or printer. A D-series device name uses the same format as a C-series device name, which is a dollar sign followed by one to seven letters or digits The first alphanumeric character in the name must be a letter. The device name can also have an optional qualifier, which is a pound sign (#) followed by one to seven letters or digits, the first of which must be a letter. Examples of valid D-series device names are:

```
$TAPE001                ! Device name
$LAZRPTR.#QWERTY        ! Device name and qualifier
\TOKYO.$TERM080         ! Node and device name
\NY.$LINEPTR.#ROOM10    ! Node, device name, and qualifier
```

Note Although logical device numbers are valid for the D-series operating system, Tandem recommends that you use a logical device name rather than a logical device number whenever possible. Logical device numbers are often unreliable (for example, with Dynamic System Configuration).

A converted process on a D-series system can identify remote device names with eight characters (seven characters after the dollar sign) on other D-series systems in a network. A process on a C-series system can identify remote device names with a maximum of six letters or digits after the dollar sign on other systems.

However, a process on a D-series system cannot identify a remote C-series volume name that has seven characters after the dollar sign. For more information about the compatibility of C-series and D-series device names, refer to Appendix C, "System Compatibility."

Table 2-2 shows a comparison of the programmatic representation of C-series and D-series device names.

Table 2-2. C-Series and D-Series Programmatic Representation of I/O Device Names

Form	C-Series Programmatic Representation		D-Series Programmatic Representation
Device name (without a specified node name) [1]	*name* [0:3]	= Device name ($ and 1 to 7 characters)	[*node-name*.]*name*[.*qualifier*] [3]
	name [4:11]	= Optional qualifier	
Network device name	*name* [0].<0:7>	= ASCII "\"	[*node-name*.]*name*[.*qualifier*] [3]
	name [0].<8:15>	= System number	
	name [1:3]	= Device name (1 - 6 characters)[2]	
	name [4:11]	= Optional qualifier	

1 The D-series operating system expands a partially qualified D-series file name using the current settings, including the node name, from the =_DEFAULTS DEFINE VOLUME attribute. A device name that does not include a node name is therefore not necessarily a local device name on a D-series system.

2 On C-series systems, a process can identify a remote device name with a maximum of six characters. A remote device name does not include a dollar sign in the programmatic representation.

3 On D-series systems, a converted process can identify a remote device name with two to eight characters including the dollar sign on other D-series systems but not on C-series systems.

New Process Identifiers

A process identifier identifies a process in a system or network. The D-series operating system uses these process identifiers:

- ☐ Process names
- ☐ Process file names
- ☐ Process descriptors
- ☐ Process handles

These identifiers are described in the following paragraphs. For compatibility with C-series systems, the D-series operating system also supports C-series process names, process file names, and process IDs.

Process Names

A process name identifies a process or process pair in a system. A D-series process name uses the same format as a C-series process name, which is a dollar sign followed by one to five letters or digits. The first character after the dollar sign must be a letter. Examples of valid process names are:

```
$SERVR
$Z146
$SPLS
$A0020
```

If a process is named, the name is assigned to the process when it is created. If you create a process using either the TACL RUN command or a process-creation procedure such as PROCESS_CREATE_ , you can specify a process name or you can request that the system generate a name for the process.

A converted process on a D-series system can identify remote processes with names that have a maximum of six characters including the dollar sign on other D-series systems in a network. A process on a C-series system can identify remote processes with names that have a maximum of five characters including the dollar sign on other systems.

However, a process on a D-series system can identify remote processes with names that have only five characters or less, including the dollar sign, on C-series systems in the network. For more information about the compatibility of C-series and D-series process names, refer to Appendix C, “System Compatibility.”

Process File Names

The D-series process file name replaces the C-series process file name. A D-series process file name is a variable-length string that specifies either a named or unnamed process (or a named process pair). The length in bytes of the string is specified as a separate integer value. You use a D-series process file name with Guardian procedures that operate on files (for example, the FILE_OPEN_ procedure).

The formats for the D-series unnamed and named process file names are described in the following paragraphs.

Process File Names for Named Processes

A D-series process file name can identify a named process. Valid examples are:

```
\LONDON.$ZAB2:4300411433 ! Process name with node and seq num
$ZSVR                    ! Process name only
\LA.$APP2.#A001.Z1       ! Process name, node, and qualifiers
```

The format of a D-series process file name for a named process is:

```
[node-name.]process-name[:sequence-number][.q1[.q2]]
```

node-name

is a variable-length string specifying the node (system) name. The name consists of a backslash (\) followed by one to seven letters or digits; the first character after the backslash must be a letter. This syntax is identical to the syntax of a system name in a C-series external-format process file name.

process-name

is a variable-length string specifying the process name. The process name consists of a dollar sign followed by one to five letters or digits; the first character after the dollar sign must be a letter.

For C-series systems connected in a network, processes with six-character names (including the dollar sign) are visible only on the local system. These processes are not visible from other C-series systems or D-series systems in the network.

For D-series systems connected in a network, processes with six-character process names (including the dollar sign) are visible from any D-series system in the network, but not from C-series systems.

Appendix C, “System Compatibility,” contains information about process-name compatibility for a network of C-series and D-series systems.

sequence-number

is a system-assigned sequence number with a maximum of 13 digits. Any leading zeros are suppressed. A colon (:) separates *process-name* from *sequence-number*.

The sequence number identifies a named process (or a named process pair) over its lifetime and can be used to detect an incorrect reference to a process. For example, a process named $B23:4321133452 terminates. A new process named $B23:4322246543 is created with the same name but with a different sequence number. An attempt to access process $B23:4321133452 fails, because this specific instance of $B23 no longer exists.

Both processes in a process pair have the same sequence number. If the primary process stops then the backup process becomes the primary process and has the

same sequence number. When the original primary process restarts, it too has the same sequence number. Only if both processes in the process pair stop is the process pair restarted with a new sequence number.

q1 and *q2*

are optional file-name qualifiers. Each qualifier can have up to eight characters. The first character of *q1* must be a pound sign (#). This syntax is identical to the syntax of C-series file-name qualifiers.

Process File Names for Unnamed Processes

A D-series process file name can identify an unnamed process. Valid examples are:

```
$:2:850:5237743650        ! CPU, PIN and seq num
\NY.$:6:200:2876540012    ! Node, CPU, PIN, and seq num
\PARIS.$:6:130:3547234520 ! Node, CPU, PIN, and seq num
```

The format of a D-series process file name for an unnamed process is:

```
[node-name.]$:cpu:pin:sequence-number
```

node-name

is a variable-length string specifying the node (system) name. The name consists of a backslash followed by one to seven letters or digits; the first character after the backslash must be a letter. This syntax is identical to the syntax for a system name in a C-series external-format process file name.

cpu

is the CPU number, which is one or two digits ranging from 0 through 15. A colon separates the dollar sign from *cpu*.

pin

is the PIN value, which is one to five digits ranging from 0 to the maximum value allowed for the CPU. A colon separates *cpu* from *pin*.

sequence-number

is a system-assigned sequence number with a maximum of 13 digits. Any leading zeros are suppressed. A colon separates *pin* from *sequence-number*.

The sequence number is unique for each process and serves the same purpose as the process creation timestamp, which is used in the C-series timestamp form of the process ID. (However, a sequence number is not a timestamp.)

The number identifies a process over its lifetime and therefore can detect incorrect references to the process. For example, an unnamed process, $:1:650:1234567890, terminates. The system creates a new process, $:1:650:9876543210, with the same

CPU number and PIN but with a different sequence number. An attempt to access process $:1:650:1234567890 fails, because the process no longer exists.

Table 2-3 shows a comparison of the programmatic representation of C-series and D-series process file names.

Table 2-3. C-Series and D-Series Programmatic Representation of Process File Names

Form	C-Series Programmatic Representation			D-Series Programmatic Representation
Timestamp unnamed process	*name*	[0].<0:1>	= 2	[*node-name*.]$:*cpu*:*pin*:*sequence-number*
	name	[0].<2:7>	= Unused	
	name	[0].<8:15>	= System number	
	name	[1:2]	= Low-order 32 bits of timestamp	
	name	[3].<0:3>	= Unused	
	name	[3].<4:7>	= CPU number	
	name	[3].<8:15>	= PIN	
	name	[4:11]	= Blank-filled	
Named process (without a specified node name) [1]	*name*	[0:2]	= Process name ($ and 1 - 5 characters)	*process-name*[:*sequence-number*] [.*qualifier1*[.*qualifier2*]] [1]
	name	[3].<0:3>	= Unused	
	name	[3].<4:7>	= CPU number	
	name	[3].<8:15>	= PIN	
	name	[4:7]	= First qualifier	
	name	[8:11]	= Second qualifier	
Network named process	*name*	[0].<0:7>	= ASCII "\"	[*node-name*.]*process-name* [:*sequence-number*][.*qualifier1*[.*qualifier2*]] [3]
	name	[0].<8:15>	= System number	
	name	[1:2]	= Process name [2] (1 - 4 characters)	
	name	[3].<0:3>	= Unused	
	name	[3].<4:7>	= CPU number	
	name	[3].<8:15>	= PIN	
	name	[4:7]	= First qualifier	
	name	[8:11]	= Second qualifier	

1 The D-series operating system expands a partially qualified D-series file name using the current settings, including the node name, from the =_DEFAULTS DEFINE VOLUME attribute. A process name that does not include a node name is therefore not necessarily a local process name on a D-series system.

2 On C-series systems, a process can identify a remote process with a name that has a maximum of four characters. A remote process name does not contain a dollar sign in the programmatic representation.

3 On D-series systems, a converted process can identify a remote process with a name that has two to six characters including the dollar sign on other D-series systems but not on C-series systems.

Process Descriptors

A process descriptor is a special case of a D-series process file name. It replaces the process ID or CRTPID for returning the identity of a process from a procedure call (such as PROCESS_CREATE_). It always contains a node name and a sequence number as well as either a process name or a dollar sign and CPU/PIN designation. A process descriptor never contains a qualifier.

The format for a process descriptor for a named process or named process pair is:

```
node-name.process-name:sequence-number
```

The format for a process descriptor for an unnamed process is:

```
node-name.$:cpu:pin:sequence-number
```

If necessary, you can use the FILENAME_EDIT_ procedure to remove the node name or sequence number from the string.

Process Handles

The D-series process handle replaces the C-series four-word process ID or CRTPID for passing information to a Guardian procedure. You use a process handle in process-control Guardian procedures such as PROCESS_STOP_ and PROCESS_ACTIVATE_ or in information procedures such as PROCESS_GETINFO_ . (Process-control procedures operate only on processes and not on files; you use a process file name for procedures that operate on files.)

A process handle is a 10-word (20-byte) structure that identifies a single named or unnamed process. For a named process pair, a process handle identifies each specific member of the pair.

Note The format for a process handle is subject to change in future releases. Your application should not try to extract information (such as a CPU number or PIN) from a process handle except by using a Guardian procedure such as PROCESSHANDLE_DECOMPOSE_.

A process handle contains the following information about a process:

- A PIN identifies the process within a specific CPU. A D-series PIN ranges from 0 through the maximum number supported by the system (except for PIN 255, which is never used in a process handle).
- A CPU number identifies the CPU in which the process is running. The CPU number ranges from 0 through 15.
- A node (or system) number identifies the node within a network. The number ranges from 0 through 254.
- A verifier or sequence number allows the system to uniquely identify a process over its lifetime.

For example, an unnamed process terminates while running in a specific CPU. The system then creates another unnamed process in the same CPU with the same PIN. The verifier number distinguishes the new process from the old process, even though both processes have the same CPU number and PIN.

- A process pair index allows the system to determine the name of a named process pair (and therefore to locate the members of a named process pair).
- A type field identifies the type of process (for example, a named or unnamed process).

Table 2-4 shows a comparison of C-series process IDs and D-series process handles.

Table 2-4. C-Series Process IDs and D-Series Process Handles

Form	C-Series Internal Representation (Process ID or CRTPID)			D-Series Representation
Timestamp unnamed process	*name*	[0].<0:1>	= 2	Process handle (20-byte structure defined by Tandem)
	name	[0].<2:7>	= Unused	
	name	[0].<8:15>	= System number	
	name	[1:2]	= Low-order 32 bits of timestamp	
	name	[3].<0:3>	= Unused	
	name	[3].<4:7>	= CPU number	
	name	[3].<8:15>	= PIN	
Local named process	*name*	[0:2]	= Process name ($ and 1 - 5 characters)	Process handle
	name	[3].<0:3>	= Unused	
	name	[3].<4:7>	= CPU number	
	name	[3].<8:15>	= PIN	
Network named process	*name*	[0].<0:7>	= ASCII "\"	Process handle
	name	[0].<8:15>	= System number	
	name	[1:2]	= Process name (1 - 4 characters)[1]	
	name	[3].<0:3>	= Unused	
	name	[3].<4:7>	= CPU number	
	name	[3].<8:15>	= PIN	

1 On C-series systems, a process can identify a remote process with a name that has a maximum of four characters. A remote process name does not contain a dollar sign in the programmatic representation.

Null Process Handle

A null process handle contains a -1 in each of its ten words. It is used for the following purposes:

- An application can specify a null process handle (for example, by using the PROECSSHANDLE_NULLIT_ procedure) as an input parameter in a D-series procedure such as PROCESS_GETINFO_ in order to get information about itself.
- Some D-series procedures return a null process handle to represent a nonexistent process (for example, a backup-process parameter when a backup process does not exist).
- An application can use a process handle as a place holder for a nonexistent process. When an application declares a process-handle variable, it can initialize the variable to a null value. If a server process monitors its openers, it can set the primary and backup process-handle fields in an opener table entry to null values as place holders for the nonexistent processes.

  If a server process uses the OPENER_LOST_ procedure, it must set any unused process-handle fields in its opener table to null values. See Section 3, “Converting TAL Applications,” for more information about defining an opener table for the OPENER_LOST_ procedure.

New System Messages

The D-series operating system provides new user-level system messages that a converted application can read from $RECEIVE. (An application that does not open $RECEIVE to request system messages need not be concerned with system messages.) An application that uses C-series Guardian procedures will also continue to receive C-series system messages.

Some D-series messages supersede one or more C-series messages, while other D-series messages support new procedures or features. For example, the D-series -101 (Process deletion) message supersedes the C-series -5 (Stop), -6 (Abend), and -2 (CPU down for a named process deletion) messages, while the D-series -109 (Nowait FILENAME_FINDNEXT_ completion) message supports the new FILENAME_FINDNEXT_ procedure.

Tandem provides structure declarations that you can use in your application to read and process system messages. Tandem uses the ZSYSDDL file to generate the ZSYSTAL, ZSYSCOB, ZSYSC, and ZSYSPAS files for TAL, COBOL85, C, and Pascal applications, respectively. To use the ZSYSDDL declarations, include the appropriate file (or sections of the file) in your source code.

For a table of C-series messages and the D-series messages that supersede them, refer to Appendix B, “System Messages.” For the format and description of all system messages, refer to the *Guardian Procedure Errors and Messages Manual*.

Improved I/O Performance

The D-series operating system improves the performance of most I/O operations by transferring data directly between an application's I/O buffers and the I/O device. A C-series I/O operation (that is, an I/O operation that uses C-series procedures) uses an intermediate buffer in the application's process file segment (PFS) for data transfers. The D-series SETMODE 72 function allows an application to specify an intermediate PFS buffer, if needed.

For more information about direct I/O transfers and the SETMODE 72 function, refer to Section 8, "Converting Other Parts of an Application."

New Distributed Systems Management (DSM) Tokens

For DSM applications that use the Event Management Service (EMS) or the Subsystem Programmatic Interface (SPI), the D-series operating system has these changes:

- ☐ New tokens define items such as the process handle and process descriptor. Tokens are also defined for new file-system errors and error lists.
- ☐ The FNAMECOMPARE, DECOMPOSE, and DECOMPOSEERROR functions are available for EMS filters.

If you are converting a DSM application, refer to Section 8, "Converting Other Parts of an Application," for more information.

New File-System Error Numbers

Table 2-5 lists the D-series file-system errors. For a description of all file-system errors, refer to the *Guardian Procedure Errors and Messages Manual*.

Table 2-5. D-Series File-System Errors

Error Number	Description
560	A calling process tried to access a subject process, but the subject process has not been converted to accept high-PIN requests.
561	The system did not recognize an item code in a list in a FILE_CREATELIST_ , FILE_ALTERLIST_ , FILE_GETINFOLIST_ , or FILE_GETINFOLISTBYNAME_ procedure.
563	The size of the output buffer in a procedure call is too small to hold the results.
564	The system detected an operation that is not allowed for this file type.
565	A server detected an inappropriate or illegal request sent from a requester.
566	A requester detected an inappropriate or illegal reply returned from a server.
590	The system detected an invalid or inconsistent parameter value.
593	An operation failed because a request was abandoned.
597	The system did not find a required item in an item list.
632	Not enough stack space was available to complete the operation.

For a description of the D-series file-system error lists, refer to Section 8, "Converting Other Parts of an Application."

New Object-File Attributes

The D-series operating system provides the following new object-file attributes:

HIGHPIN	Use this attribute to tell the system that the program can be run at a high PIN.
HIGHREQUESTERS and RUNNAMED	Use these attributes to allow communication between converted and unconverted applications . These attributes can remove the need to convert certain processes.

You set these object-file attributes as follows:

- ☐ When you compile your program, use a compiler directive in the source file or as a compiler option when you start the compiler.
- ☐ After you have compiled your program, use the Binder program.

For information about setting these attributes for TAL, COBOL85, C, and Pascal applications, refer to Sections 3 through 6, respectively. For information about setting setting the HIGHPIN directive using TACL, see Section 7, “Converting TACL Programs.”

The following paragraphs describe these object-file attributes.

The HIGHPIN Attribute

Use the HIGHPIN attribute to indicate that it is acceptable to run the object file as part of a process that runs at a high PIN. For example, the program does not contain calls to MYPID.

For a process to run at a high PIN, all the following are required:

- ☐ All participating object files (including the user library file, if present) must have the HIGHPIN attribute set.
- ☐ The creating process agrees that it is acceptable to run the process at a high PIN. To do this, the creating process must create the new process using the PROCESS_CREATE_ procedure rather than NEWPROCESS[NOWAIT] and must not demand a low PIN for the process.
- ☐ Either of the following is true:
  - ☐ The inherited force-low characteristic for the new process is not set
  - ☐ The inherited force-low characteristic for the new process is set, and the creating process overrides it

  The inherited force-low characteristic is a mechanism that, for compatibility reasons, forces descendents of low-PIN processes to run at a low PIN, unless the creating process explicitly overrides the mechanism. See Appendix C, “System Compatibility,” for details.
- ☐ A high PIN must be available.

The HIGHREQUESTERS Attribute

The HIGHREQUESTERS attribute allows an unconverted server process to support high-PIN requesters if the server meets certain requirements; that is, if the server does not examine its openers. This attribute gives you the option of not converting the server process, but you must be sure that the server process allows you to do this.

Appendix C, "System Compatibility," provides additional information on the use of the HIGHREQUESTERS attribute.

The RUNNAMED Attribute

The RUNNAMED attribute causes a process to run as a named process even if the creator does not provide a name. This attribute allows your process to run at a high PIN and be opened by an unconverted process. Again, this option can work only if the process does not examine its openers.

Conversion Considerations and the Common Run-Time Environment (CRE)

The C-series C, COBOL85, FORTRAN, Pascal, and TAL programming languages each have their own run-time environments defined by their respective run-time libraries. These language-specific run-time environments are different from one another and are often incompatible. Mixed-language programs running in these environments are limited in their ability to use all of the features of each language and to share data between routines written in different languages. This incompatibility severely limits the potential for creating useful mixed-language programs.

The Common Run-Time Environment (CRE) eliminates this problem. The CRE coordinates many run-time tasks on behalf of the run-time libraries, thus providing a common environment for routines in a program, regardless of language. The CRE provides services that significantly enhance your ability to create mixed-language programs. These services include:

- ☐ Shared access to standard files (standard input, standard output, standard log)
- ☐ Shared access to a common run-time heap
- ☐ Management of $RECEIVE
- ☐ Management of process pairs
- ☐ Uniform handling of exceptions
- ☐ Uniform form and content of diagnostic messages

Routines in a program that runs in the CRE call language-specific run-time libraries as in prior releases. However, these language-specific run-time libraries now call the CRE library for some operations. Programs that run in the CRE can also make explicit calls to CRE library routines. The CRE library replaces large portions of each language's run-time library; it does not replace Guardian procedures. The CRE allows all routines in a program, regardless of language, to use their own run-time libraries as well as CRE library routines and Guardian procedures.

All C-series compilers generate programs that run in a language-specific run-time environment. The D-series C and Pascal compilers always generate programs that run in the CRE. You can specify a compiler directive to instruct the D-series COBOL85, FORTRAN, and TAL compilers to generate programs that run in the CRE.

For programs that contain C, COBOL85, FORTRAN, and Pascal routines but do not contain TAL routines, no changes are required to run in the CRE. There are no changes in execution, except perhaps the text and format of run-time diagnostic messages in C and Pascal, which come from the CRE error-reporting routines.

For programs that contain TAL procedures, you must change calls from Guardian procedures to CRE library routines to perform actions on those objects, such as the standard files, that you want to share or coordinate with other languages. TAL routines can continue to use Guardian procedure calls for resources that are not shared. In addition, TAL routines cannot directly manipulate the upper 32K of the run-time heap. The CRE uses this area to store its data.

3 Converting TAL Applications

A TAL program can run at a low PIN under the D-series operating system without any changes. However, for a TAL program to use the extended features of the D-series operating system, specific parts of it must be converted. The topics in this section are:

- ☐ Converting basic elements of a TAL program, such as using the EXTDECS and ZSYSTAL files, declaring variables, and running the TAL compiler
- ☐ Converting a TAL program to run at a high PIN
- ☐ Converting a TAL program to create and manage a high-PIN process
- ☐ Converting a TAL requester to communicate with a high-PIN server
- ☐ Converting a TAL program to allow a high-PIN creator
- ☐ Converting a TAL server to communicate with a high-PIN requester

The topics in this section can also apply to a COBOL85, C, or Pascal application that calls Guardian procedures. For information about calling Guardian procedures from COBOL85, C, or Pascal, refer to the specific language manual. You cannot call Guardian procedures directly from TACL, but you can access them indirectly through built-in functions; for details on TACL built-in functions, see the *TACL Reference Manual* for details on TACL built-in functions.

Section 8, "Converting Other Parts of an Application," also contains information about converting other parts of a TAL application. For additional information about TAL, refer to the *Transaction Application Language (TAL) Reference Manual* and the *Transaction Application Language (TAL) Programmer's Guide.*

Converting Basic Elements of a TAL Program

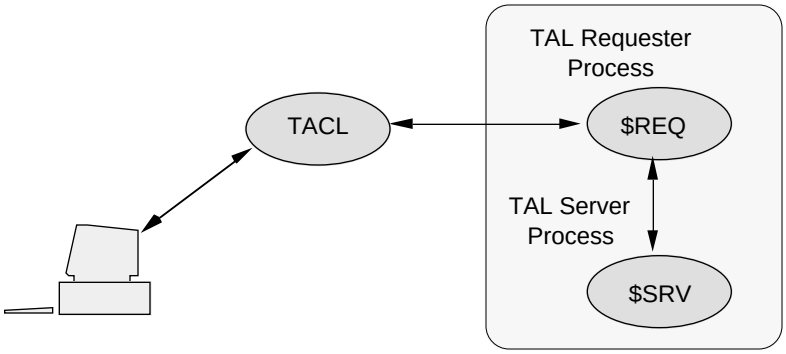
This subsection describes conversion that applies to all TAL programs you need to convert to run under the D-series operating system, irrespective of what the program does. Later subsections describe how to convert specific actions that your program might take.

This subsection discusses the following topics:

- ☐ Using source declarations from the EXTDECS file
- ☐ Using source declarations from the ZSYSTAL file
- ☐ Declaring and using variables for high PINs, file system error numbers, file names, and process identifiers
- ☐ Running the TAL compiler

Figure 3-1 shows a typical application. The box shows which processes this part of the conversion applies to. Converting basic elements of a TAL program applies to both of these processes.

Figure 3-1. Converting Basic Elements of a TAL Program



Using the EXTDECS Declarations

Your existing program should use the SOURCE compiler directive to specify the Guardian procedure declarations in the $SYSTEM.SYSTEM.EXTDECS0 file. For example, this SOURCE directive specifies C-series procedures:

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (OPEN,
?                                READX,
?                                WRITEX,
?                                CLOSE)
```

Convert your SOURCE directive to specify declarations from the D-series EXTDECS0 file. Use the procedure names for any D-series enhanced procedures your program calls:

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (FILE_OPEN_,
?                                READX,
?                                WRITEX,
?                                FILE_CLOSE_)
```

Specify only the D-series EXTDECS file (EXTDECS0). The C-series compatible declarations are also available in this file.

Note Although the file names are identical in the two examples shown here, they actually identify different files because EXTDECS0 always names the most recent EXTDECS file.

Using the ZSYSTAL Declarations

Tandem provides source declarations of TAL declarations and structures for Guardian procedures and system messages in the ZSYSTAL file. This file is typically found on the $SYSTEM.ZSYSDEFS subvolume. Contact your system manager for the location of the file on your system.

To use these declarations, include them with your source code using the SOURCE compiler directive. For example, this SOURCE directive includes the entire ZSYSTAL file:

```
?SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
```

The ZSYSTAL file is divided into sections, which allows you to include only the sections your program actually needs. For example, this SOURCE directive includes only the process-creation and system-message constant declarations:

```
?SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL (PROCESS^CONSTANT,
?                                 SYSTEM^MESSAGES^CONSTANT)
```

To print a listing of the ZSYSTAL file to check the declarations that are available for your program, use the FUP COPY command:

```
10> FUP COPY $SYSTEM.ZSYSDEFS.ZSYSTAL, $s.#lineptr
```

Declaring and Using Programming Variables

When converting your program to use the D-series enhanced interface, there are several variables that you might need to add or modify. These variables include:

- ☐ Variables that contain a PIN
- ☐ File-system error numbers
- ☐ Guardian file names, including disk file names, device names, and process file names
- ☐ Process identifiers, including process IDs, process handles, and process descriptors

The following paragraphs describe how to declare these entities to the D-series enhanced interface.

Declaring CPU and PIN Variables

Your existing program might declare either a 16-bit integer variable for both the CPU and PIN values or an 8-bit variable for a PIN value:

```
STRING primary^cpu,       ! CPU of primary process.
       primary^pin;       ! PIN of primary process.

INT    cpu^and^pin;       ! CPU and PIN values.
```

Declare all PIN values, including backup-process PINs, as 16-bit integer variables. For CPU and PIN values, declare each item as a separate 16-bit integer variable:

```
INT    primary^cpu,       ! CPU of primary process.
       primary^pin;       ! PIN of primary process.
```

Declaring and Checking File-System Error Numbers

You might need to convert the parts of your program that declare and check file-system error variables. For example, your program might declare a STRING variable for a file-system error number:

```
STRING fs^error^number;
```

Declare a file-system error number as a 16-bit integer variable:

```
INT fs^error^number;
```

Your program might also include procedures that assume a maximum value for a file-system error number (for example, 255). A D-series file-system error number can be a maximum of 16 bits. Therefore, to make sure that your procedures do not exclude any new error numbers, you should allow for up to 5 decimal digits for error numbers.

Also, because Tandem might define additional error numbers in future releases, do not consider currently undefined numbers as invalid.

For a list and description of all file-system error numbers, refer to the *Guardian Procedure Errors and Messages Manual*.

Using Guardian File Names

Guardian file names include disk file names, device names (such as a printer or terminal name), and process file names. You might need to convert the parts of your program that declare and use file-name variables as described in the following paragraphs.

The C-series-compatible interface uses file names that have an internal format, have a fixed length of 12 words, and are fully qualified. Unconverted programs that interact with users typically:

1. Accept an external form of file name.
2. Convert the file name to uppercase.
3. Pass the uppercase file name to the FILENAMEEXPAND procedure, which returns the internal format.
4. Pass the result to a C-series-compatible system procedure.

Programs that use the D-series enhanced interface can, in most cases, take a name and pass it directly to the Guardian procedure, which handles case shifting and defaulting. If file-name manipulation is necessary, the D-series enhanced interface has procedures to do the manipulation for you.

Disk File Names. Your existing program might declare a Guardian disk-file-name variable. The largest D-series disk file name are:

For permanent files	35 bytes (one byte larger than the external form of a C-series network file name)
For temporary files	26 bytes (4 bytes larger than the external form of a C-series network file name)

When accessing Guardian disk files on remote D-series systems in a network, a converted program can use a D-series network disk file name with an eight-character volume name (one to seven characters after the dollar sign). A C-series network disk file name allows a maximum of six characters after the dollar sign in the volume name.

Therefore, if you want to take advantage of this extension, you must declare your network file-name variables large enough to include this extra character. To ensure that your declaration is long enough, use the ZSYS^VAL^LEN^FILENAME LITERAL (47 bytes) from the ZSYSTAL file. For example:

```
STRING .employee[0:34] :=
          ["\newyork.$payroll.july1990.employee"];

STRING .managers[0:ZSYS^VAL^LEN^FILENAME-1] :=
          ["\newyork.$disk4.level2.managers    "];
```

Device Names. Your existing program might declare a variable for a Guardian device name. The largest D-series device names are:

Device name without a node name or qualifier	8 bytes (same as a C-series name)
Device name without a node name but with a qualifier	17 bytes (same as a C-series name)
Network device name without a qualifier	17 bytes (one byte larger than a C-series name)
Network device name with a qualifier	26 bytes (one byte larger than a C-series name)

When accessing devices on remote D-series systems in a network, a converted program can use an eight-character network device name (one to seven characters after the dollar sign). A C-series network device name allows a maximum of six characters after the dollar sign.

Therefore, you might need to declare your network device names large enough to include this extra character. To ensure that your declaration is long enough, use the ZSYS^VAL^LEN^FILENAME LITERAL (47 bytes) from the ZSYSTAL file. For example:

```
! Network device name without a qualifier

STRING .device^name[0:16] := ["\hamburg.$term001"];

! Network device name with a qualifier

STRING .network^device^name[0:ZSYS^VAL^LEN^FILENAME-1] :=
               ["\hamburg.$lineptr.#room025"];
```

Process File Names. Your existing program might declare a variable for a C-series process file name. The D-series operating system uses D-series process file names instead of C-series process file names. Use C-series process file names for compatibility with unconverted C-series procedures.

The following example shows a declaration for a D-series process file name that makes use of the ZSYSTAL file:

```
! D-series process file name

STRING .proc^filename[0:ZSYS^VAL^LEN^FILENAME-1] :=
                ["\east.$S.#wide"];
```

Declaring Process Identifiers

Your existing program might declare a four-word process-ID variable to identify a process:

```
INT process^id[0:3];
```

Convert the process-ID variable declaration to a process-handle variable for process-control operations or to a process-descriptor variable for returning information from a Guardian procedure. Use a process-ID variable for compatibility with unconverted C-series procedures.

A process handle is a 10-word (20-byte) fixed-length structure. A process descriptor is a specific form of the D-series process file name that always includes a node name and sequence number. The ZSYSTAL file contains declarations that you can use for declaring process-handle and process-descriptor variables. For example:

```
! process handle

INT .my^process^handle[0:ZSYS^VAL^PHANDLE^WLEN-1];

! process descriptor

STRING .proc^desc[0:ZSYS^VAL^LEN^PROCESSDESCR-1];
```

Avoiding Subvolume Defaulting

Your existing program might use subvolume defaulting to represent a Guardian disk file name in the form *volume.file-id*. For example:

```
STRING .disk^file[0:16] := ["$diskvol.filename"];
```

If you are using the D-series programmatic interface, avoid subvolume defaulting. If a file name requires the volume name, also include the subvolume name:

```
STRING .disk^file[0:ZSYS^VAL^LEN^FILENAME-1] :=
                              ["$diskvol.subvol.filename  "];
```

Running the TAL Compiler

When you start the TAL compiler using the TACL RUN command, TACL calls the PROCESS_CREATE_ procedure to create the TAL compiler process and the resulting BINSERV and SYMSERV processes at low PINs.

To run the TAL compiler process and the BINSERV and SYMSERV processes at high PINs, you must use the Binder program to set the HIGHPIN object-file attribute to ON in the TAL compiler object file (provided you have the proper authority to change this file):

```
@CHANGE HIGHPIN ON IN ztal
```

TACL then runs the TAL compiler and the BINSERV and SYMSERV processes at high PINs if they are available. For more information about TACL, refer to the *TACL Reference Manual*.

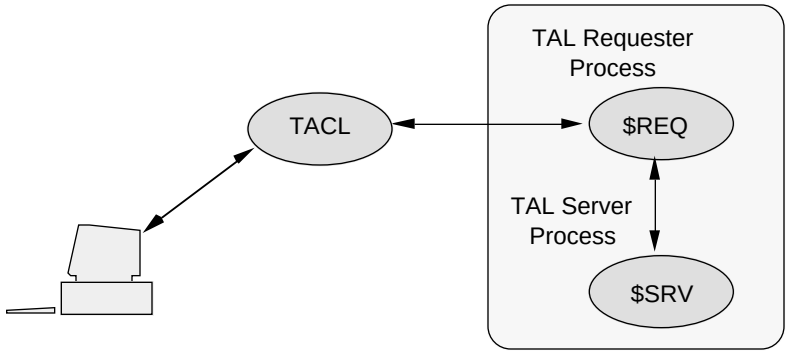
Using the Binder With Converted Object Files

You cannot bind modules that have been compiled with the D-series compiler with modules that have been compiled with a C-series compiler. If you compile any module with the D-series compiler, you must also recompile any other modules that you bind it with.

Converting a TAL Program to Run at a High PIN

This subsection describes how to convert a TAL program to run at a high PIN under the D-series operating system. Figure 3-2 shows a typical application. The box shows which processes this part of the conversion applies to. Converting a TAL program to run at a high PIN applies to both of these processes.

Figure 3-2. Converting a TAL Program to Run at a High PIN



To convert your program to run at a high PIN, you must:

- ☐ Set the HIGHPIN object-file attribute (which informs the system that your program can run at a high PIN)
- ☐ Make sure that each library file your program uses also has its HIGHPIN object-file attribute set (and is capable of running at a high PIN)
- ☐ Declare PIN variables large enough to hold high-PIN values
- ☐ Convert MYPID procedure calls into calls to the PROCESS_GETINFO_ and PROCESSHANDLE_DECOMPOSE_ procedures

Setting the HIGHPIN Object-File Attribute

The HIGHPIN object-file attribute informs the system that the object file can run at a high PIN. You set the HIGHPIN object-file attribute either during compilation using a compiler directive or after compilation using the Binder program.

To set the attribute when you compile your program, specify the HIGHPIN directive in your source code or as a compiler option in the TACL RUN command for the TAL compiler. The BINSERV program then sets the HIGHPIN attribute in the object file. An example of this directive in your source file is:

```
?HIGHPIN
```

An example of this directive as a compiler option is:

```
10> TAL /IN talsrc, OUT $s.#tallst, NOWAIT/ talobj; HIGHPIN
```

You need to specify the HIGHPIN directive only once during a compilation. The TAL compiler ignores redundant HIGHPIN directives.

If you do not set the HIGHPIN attribute when you compile your program, you can set it after compilation using the Binder program.

If you are binding more than one object file into a single target object file, the Binder program sets the HIGHPIN object-file attribute in the target file only if all constituent files have the HIGHPIN object-file attribute set. If necessary, use the Binder CHANGE command to set the attribute in the target object file:

```
@CHANGE HIGHPIN ON IN talobj
```

Using a Library File

If your program uses a library file, the library file must also have the HIGHPIN object-file attribute set. To determine the current setting of the HIGHPIN attribute for the library file, use the Binder SHOW command:

```
@SHOW SET HIGHPIN FROM libfile
```

If necessary, set this attribute as described in the previous step (provided the library file has been converted to support a high-PIN process).

Declaring 16-Bit CPU and PIN Variables

As described earlier under “Converting Basic Elements of a TAL Program,” you need to provide a 16-bit variable for the CPU value and a 16-bit variable for the PIN value.

Your existing program might declare either a 16-bit integer variable for both the CPU and PIN values or an 8-bit variable for a PIN value:

```
STRING primary^cpu,       ! CPU of primary process.
       primary^pin;       ! PIN of primary process.

INT    cpu^and^pin;       ! CPU and PIN values.
```

Declare all PIN values, including backup-process PINs, as 16-bit integer variables. For CPU and PIN values, declare each item as a separate 16-bit integer variable:

```
INT    primary^cpu,       ! CPU of primary process.
       primary^pin;       ! PIN of primary process.
```

Calling the MYPID Procedure

If a high-PIN process calls MYPID, a trap condition occurs; MYPID cannot return 16 bits to an 8-bit field. Your existing program might call the MYPID procedure to obtain its CPU and PIN values:

```
INT cpu^and^pin;          ! CPU and PIN values.

...

cpu^and^pin := MYPID;     ! Return the CPU and PIN values.
```

Convert MYPID calls into PROCESSHANDLE_DECOMPOSE_ procedure calls except when MYPID is called within a SETMODE function 11 call as described under "Using MYPID in a SETMODE (Function 11) Procedure Call," the next subsection.

The PROCESSHANDLE_DECOMPOSE_ procedure requires a process handle as an input parameter. If you do not know the process handle of your process, call the PROCESSHANDLE_GETMINE_ procedure. Then pass the results to PROCESSHANDLE_DECOMPOSE_ , which returns the CPU and PIN values as separate integer values. For example:

```
INT cpu^number,                          ! CPU.
    pin^number,                          ! PIN.
    process^handle[0:ZSYS^VAL^PHANDLE^WLEN-1];
                                         ! Process handle.

...

! Obtain the caller's process handle.

error := PROCESSHANDLE_GETMINE_(process^handle);

 ... ! Check the error return value.

! Return the caller's CPU and PIN values.

error := PROCESSHANDLE_DECOMPOSE_(process^handle,
                                  cpu^number,
                                  pin^number);
```

Your program might also specify MYPID as a parameter to another procedure call (for example, GETCRTPID or PROCESSINFO). See "Getting Information About a High-PIN Process" later in this section for information about these procedures.

Using MYPID in a SETMODE (Function 11) Procedure Call

Your existing program might also attempt to establish break ownership using a call to the MYPID procedure in a SETMODE (function 11) procedure call as shown in the following example:

```
LITERAL set^break^owner = 11;

...

CALL SETMODE (terminal^number,
              set^break^owner,
              MYPID,      ! Call MYPID to set parameter-1.
              normal^mode,
              previous^owner);
```

You are not required to set *parameter-1* to your CPU and PIN values. Instead, convert your program to set *parameter-1* to any positive value:

```
LITERAL break^flag = 1;

...

CALL SETMODE (terminal^number,
              set^break^owner,
              break^flag,         ! Dummy value.
              normal^mode,
              previous^owner);
```

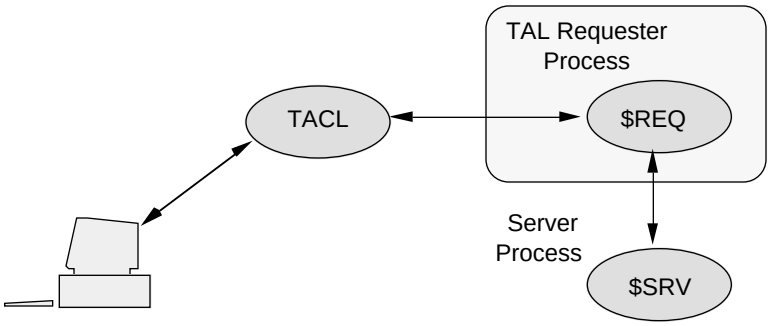
For more information about function 11 of the SETMODE procedure, refer to Section 8, “Converting Other Parts of an Application.”

Creating and Managing a High-PIN Process

This subsection describes how to convert a program to create and manage a new process using the D-series programmatic interface. The process you create must already be converted to run at a high PIN as described under "Converting a TAL Program to Run at a High PIN" earlier in this section.

Figure 3-3 shows the processes involved in converting this part of an application. The steps described in this subsection apply to the requester process $REQ, which creates the high-PIN server process $SRV.

Figure 3-3. Converting a TAL Program to Create and Manage a High-PIN Process



This subsection covers these topics:

- ☐ Creating a high-PIN process in a waited or nowait manner using the PROCESS_CREATE_ procedure
- ☐ Specifying a process name and other options using PROCESS_CREATE_
- ☐ Using the process-name, process-descriptor, and process-handle parameters from a PROCESS_CREATE_ procedure call
- ☐ Creating a low-PIN process using PROCESS_CREATE_
- ☐ Managing a high-PIN process, including activating, suspending, stopping, and abending the process, as well as invoking Inspect or Debug for the process
- ☐ Getting information about a high-PIN process
- ☐ Setting process attributes for a high-PIN process

Creating a High-PIN Process

To convert a program to create a high-PIN process, follow the steps in this subsection. The ZSYSTAL file contains LITERAL declarations that you can use with the PROCESS_CREATE_ procedure. This subsection describes:

- ☐ How to programmatically create high-PIN processes in a waited manner
- ☐ How to programmatically create high-PIN processes in a nowait manner

For information on how to interactively create a high-PIN process using TACL, refer to Section 7, "Converting TACL Programs."

To avoid running out of low PINs, we usually recommend running application processes at high PINs. However, because the rules for communicating between processes make it impossible for an unconverted low-PIN process to communicate with a high-PIN process, you sometimes must run a given process at a low PIN.

The inherited force-low characteristic of a process usually causes a process with ancestors on C-series systems to run at a low PIN to enable the ancestor to open the process and communicate with it. You have the option to override the inherited force-low characteristic.

Appendix C, "System Compatibility," explains the rules for communicating with high-PIN and low-PIN processes among C-series systems and D-series systems, including details on the inherited force-low characteristic.

Creating a High-PIN Process in a Waited Manner

Your existing program might call the NEWPROCESS procedure to create a process in a waited manner:

```
CALL NEWPROCESS (program^file,
                 priority,
                 memory^pages,
                 processor,
                 process^id,
                 error);
```

To convert your program to create a high-PIN process in a waited manner, you must make the following changes in your source code:

1. Use the PROCESS_CREATE_ procedure rather than the NEWPROCESS procedure. NEWPROCESS always creates a low-PIN process.
2. Each file-name parameter must be a variable-length string with its length specified as a separate integer value. Set the *program^file* parameter and, if needed, the *library^file*, *swapfile*, and *ext^swapfile* parameters and their respective lengths.
3. Set the *nowait^tag* parameter to -1D, or omit this parameter from the call.
4. If you use the *create^options* parameter, make sure that bit 15 is zero. If you do not use *create^options*, this bit is zero by default. If this bit is not zero, the system creates the process at a low PIN.

5. Set *create^options*.<10> to 1 if you want the process to run at a high PIN irrespective of process ancestry. Set *create^options*.<10> to zero (the default value) if you want process ancestry to influence whether the process runs at a high PIN or a low PIN.

In the following example, PROCESS_CREATE_ creates a high-PIN process in a waited manner, assuming the inherited force-low flag is not set. The *nowait^tag* and *create^options* parameters are omitted from the call:

```
error := PROCESS_CREATE_(program^file:pf^length,
                         ! library^file:lf^length  ! ,
                         ! swapfile:sf^length      ! ,
                         ! ext^swapfile:esf^length ! ,
                         priority,
                         cpu^number,
                         process^handle,
                         error^detail);
```

When you call PROCESS_CREATE_ in a waited manner, you receive the results directly in the PROCESS_CREATE_ output parameters. The system returns any errors in the returned value *error* and in the *error^detail* (if needed) output parameter.

Creating a High-PIN Process in a Nowait Manner

Your existing program might call the NEWPROCESSNOWAIT procedure to create a process in a nowait manner:

```
CALL NEWPROCESSNOWAIT(program^file,
                      priority,
                      memory^pages,
                      processor,
                      ! process^id ! ,
                      error);
```

To convert your program to create a high-PIN process in a nowait manner, you must make the following changes in your source code:

1. Use the PROCESS_CREATE_ procedure rather than the NEWPROCESSNOWAIT procedure. NEWPROCESSNOWAIT always creates a low-PIN process.
2. Each file-name parameter must be a variable-length string with its length specified as a separate integer value. Set the *program^file* parameter and, if needed, the *library^file*, *swapfile*, and *ext^swapfile* parameters and their respective lengths.
3. Set the *nowait^tag* parameter to a value other than -1D.
4. If you use the *create^options* parameter, make sure that bit 15 is zero. If you do not use *create^options*, this bit is zero by default. If this bit is not zero, the system creates the process at a low PIN.

5. Set *create^options*.<10> to 1 if you want the process to run at a high PIN irrespective of process ancestry. Set *create^options*.<10> to zero (the default value) if you want process ancestry to influence whether the process runs at a high PIN or a low PIN.

In this example, PROCESS_CREATE_ creates a high-PIN process in a nowait manner, assuming the inherited force-low flag is not set. The *nowait^tag* parameter is set to a value other than -1D:

```
error := PROCESS_CREATE_(program^file:pf^length,
                         ! library^file:lf^length     ! ,
                         ! swapfile:sf^length         ! ,
                         ! ext^swapfile:esf^length    ! ,
                         priority,
                         cpu^number,
                         process^handle,
                         error^detail,
                         ! name^option                ! ,
                         ! proc^name:proc^name^length ! ,
                         ! process^desc:max^length    ! ,
                         ! process^desc^length        ! ,
                         nowait^tag);   ! Value is -1D.
```

When you call PROCESS_CREATE_ in a nowait manner, you receive the results as follows:

☐ System message -102 (PROCESS_CREATE_ completion)

If the system creates the process successfully, or if the system initiates the process creation and an error occurs, the system returns the results to $RECEIVE in system message -102. You read this message from $RECEIVE using the READX or READUPDATEX procedure (provided you have already opened $RECEIVE to receive system messages).

This message is analogous to system message -12 (NEWPROCESSNOWAIT completion). For the description and format of all system messages, refer to the *Guardian Procedure Errors and Messages Manual*.

☐ The returned value *error* and the *error^detail* output parameter

If the system successfully initiates the creation, the returned value *error* is zero.

If the system cannot initiate the process creation, the system returns the results in the returned value *error* and the *error^detail* output parameter and does not send system message -102 to $RECEIVE.

For example, if the system encounters a parameter error, then *error* is 2 and the *error^detail* parameter contains the ordinal number of the first parameter that the system detected as causing an error.

Specifying a Process Name Using PROCESS_CREATE_

Your existing program might call the CREATEPROCESSNAME or CREATEREMOTENAME procedure to obtain a process name to use in the NEWPROCESS[NOWAIT] procedure call:

```
CALL CREATEPROCESSNAME (process^name);

CALL NEWPROCESS (filenames,
                 priority,
                 memory^pages,
                 processor,
                 process^id,
                 error,
                 process^name);
```

Convert your program to generate a process name in one of the following ways:

- ☐ Use the PROCESS_CREATE_ procedure to allow the system to generate a name. You do not need to use a separate procedure to obtain the process name.
- ☐ Supply the name yourself (or have the user supply the name).
- ☐ Use the PROCESSNAME_CREATE_ procedure to supply a system-generated name then pass the name to PROCESS_CREATE_.

A system-generated process name has the form $X*ddd*, $Y*ddd*, or $Z*aaa*, where *d* is any numeric character and *a* is any alphanumeric character.

System-Generated Name Without Using PROCESSNAME_CREATE_

To have the system generate a name for a new process, follow these steps:

1. Set the *name^option* parameter to the LITERAL ZSYS^VAL^PCREATEOPT^NAMEDBYSYS.
2. Omit the *name* parameter, or set the *name^length* parameter to zero.
3. Set any other PROCESS_CREATE_ parameters as needed and call the procedure.
4. Check the results as follows:
   - ☐ For a process created in a waited manner, the system returns a process descriptor and its length in the *process^desc* and *process^desc^length* output parameters.
   - ☐ For a process created in a nowait manner, the system returns a process descriptor in the system message -102. Read this message from $RECEIVE using the READ or READUPDATE procedure.

In this example, PROCESS_CREATE_ creates a high-PIN process in a waited manner. The system-supplied name is returned in the process^desc parameter:

```
name^option := ZSYS^VAL^PCREATEOPT^NAMEDBYSYS;
max^length  := ZSYS^VAL^LEN^PROCESSDESCR;
error := PROCESS_CREATE_(program^file:pf^length,
                         ! library^file:lf^length     ! ,
                         ! swapfile:sf^length         ! ,
                         ! ext^swapfile:esf^length    ! ,
                         priority,
                         cpu^number,
                         process^handle,
                         error^detail,
                         name^option,
                         ! proc^name:proc^name^length ! ,
                         process^desc:max^length,
                         process^desc^length);
```

User-Specified Name

Alternatively, you can specify a name for the new process when calling PROCESS_CREATE_ by following these steps:

1. Set the *name^option* parameter to the LITERAL ZSYS^VAL^PCREATEOPT^NAMEINCALL.
2. Set the *name* parameter to the process name. This name must be a variable-length file-name string.
3. Set the *name^length* parameter to the length in bytes of the process name.
4. Set any other PROCESS_CREATE_ parameters as needed and call the procedure. If the operation is successful, the system creates the process with the name you specified in Step 2.
5. Check the results as described in Step 4 under "System-Generated Name," earlier in this section.

In this example, PROCESS_CREATE_ creates a high-PIN process in a waited manner. The user supplies the name and name length:

```
name^option := ZSYS^VAL^PCREATEOPT^NAMEINCALL;
proc^name ':=' "$MYPROC.#first" -> @S^PTR;
proc^name^length := @S^PTR '-' @proc^name;
max^length  := ZSYS^VAL^LEN^PROCESSDESCR;
error := PROCESS_CREATE_(program^file:pf^length,
                         ! library^file:lf^length     ! ,
                         ! swapfile:sf^length         ! ,
                         ! ext^swapfile:esf^length    ! ,
                         priority,
                         cpu^number,
                         process^handle,
                         error^detail,
                         name^option,
                         proc^name:proc^name^length,
                         process^desc:max^length,
                         process^desc^length);
```

Note If the RUNNAMED object-file attribute is set for the program file, the D-series operating system generates a name even if the PROCESS_CREATE_ *name^option* parameter is zero or omitted.

System-Generated Name Using PROCESSNAME_CREATE_

Alternatively, you can let the PROCESSNAME_CREATE_ procedure create a process name which you later pass to the PROCESS_CREATE_ procedure. The following steps show how:

1. Call the PROCESSNAME_CREATE_ procedure to obtain a system-generated name and its length.
2. Set the *name^option* parameter of the PROCESS_CREATE_ procedure to the LITERAL ZSYS^VAL^PCREATEOPT^NAMEINCALL.
3. Set the *name* parameter of PRCOESS_CREATE_ to the process name returned by the PROCESSNAME_CREATE_ procedure.
4. Set the *name^length* parameter of PRCOESS_CREATE_ to the length in bytes of the process name.
5. Set any other PROCESS_CREATE_ parameters as needed and call the procedure. If the operation is successful, the system creates the process with the name you specified in Step 2.
6. Check the results as described in Step 4 under “System-Generated Name,” earlier in this section.

In this example, PROCESS_CREATE_ creates a high-PIN process in a waited manner. The system provides the process name and length using the PROCESSNAME_CREATE_ procedure:

```
max^length  := ZSYS^VAL^LEN^PROCESSDESCR;
error := PROCESSNAME_CREATE_(proc^name:max^length,
                             proc^name^length);

...
name^option := ZSYS^VAL^PCREATEOPT^NAMEINCALL;
error := PROCESS_CREATE_(program^file:pf^length,
                         ! library^file:lf^length     ! ,
                         ! swapfile:sf^length         ! ,
                         ! ext^swapfile:esf^length    ! ,
                         priority,
                         cpu^number,
                         process^handle,
                         error^detail,
                         name^option,
                         proc^name:proc^name^length,
                         process^desc:max^length,
                         process^desc^length);
```

Using the Process Name From PROCESS_CREATE_

After calling PROCESS_CREATE_ , you might want to save the process name (if one exists) from either the *process^desc* output parameter or system message -102 for later use in your program. Save this name immediately after you receive it, because:

- If you don't save the name and you need the name later, you must convert the process handle to a process name using the PROCESSHANDLE_TO_FILENAME_ procedure. If the process is running on a remote system, using PROCESSHANDLE_TO_FILENAME_ implicitly sends a message to the remote system. Your program must wait until the operation finishes.
- If the process handle represents a named process (or process pair), you cannot determine the process name from the process handle after the named process (or process pair) has terminated.

Using the Process Handle and Process Descriptor From PROCESS_CREATE_

After calling the NEWPROCESS[NOWAIT] procedure, your existing program might use the four-word *process^id* output parameter directly in a file-system procedure such as an OPEN call. For example:

```
CALL NEWPROCESS (filenames,
                 priority,
                 memory^pages,
                 processor,
                 process^id,
                 error,
                 process^name);

...

CALL OPEN (process^id,
           file^number);
```

Instead of a four-word process ID, PROCESS_CREATE_ returns these parameters for the new process:

- A process handle, which you can use with process-control procedures such as PROCESS_STOP_ and PROCESS_ACTIVATE_ and with status-monitoring procedures such as OPENER_LOST_ and CHILD_LOST_.
- A process descriptor, which always includes the node name and a system-assigned sequence number. The *process^desc^length* parameter contains the length in bytes of the process descriptor.

Use the process descriptor directly in the FILE_OPEN_ procedure call to open the process:

```
error := PROCESS_CREATE_(program^file:pf^length,
                         ! library^file:lf^length  ! ,
                         ! swap^file:sf^length     ! ,
                         ! ext^swapfile:esf^length ! ,
                         ! priority               ! ,
                         ! cpu^number             ! ,
                         process^handle,
                         error^detail,
                         ! name^option:length       ! ,
                         process^desc:max^length,
                         process^desc^length);

...

error := FILE_OPEN_(process^desc:process^desc^length,
                    file^number);
```

If you do not want the system name or sequence number for other uses of the process descriptor, call the FILENAME_DECOMPOSE_ procedure to remove either or both of these parts. For example, you might want to remove the sequence number before you display the process descriptor.

Specifying Other PROCESS_CREATE_ Options

Most PROCESS_CREATE_ parameters are analogous to the NEWPROCESS or NEWPROCESSNOWAIT parameters. The following features are specific to PROCESS_CREATE_:

- Swap file for the extended data segment

  You can specify a swap file for the extended data segment of the new process. This swap file must be on the same system as the new process. The swap-file-name parameter must be a variable-length string with its length specified by a separate integer parameter.

- Saved DEFINEs

  You can specify a set of saved DEFINEs for the new process. You must have previously saved these DEFINEs in a buffer using one or more calls to the DEFINESAVE procedure. The DEFINE buffer must be a variable-length string with its length specified by a separate integer parameter.

  Thus, the creator process can give the new process its own DEFINEs or a set of saved DEFINEs (or both).

- Process-deletion message recipient

  You can specify that the process-deletion message (system message -101) from the process you create be delivered according to D-series rules or according to C-series rules. According to D-series rules, the process-deletion message is delivered only to the specific instance of the process that created the terminating process.

According to C-series rules, the process-deletion message is delivered to whatever process has the same name as the process that created the terminating process.

For a description of these features, refer to the *Guardian Procedure Calls Reference Manual*.

Creating a Low-PIN Process

Your program might need to create a new process that must run at a low PIN. For example, a process must run at a low PIN to access files on a remote C-series system.

To create a low-PIN process, call the PROCESS_CREATE_ procedure with the *create^options*.<15> bit set to 1. If you use the ZSYSTAL file, set the *create^options* parameter to ZSYS^VAL^PCREATOPT^LOWPIN. The system creates the new process at a low PIN regardless of the HIGHPIN attribute setting for the program object file:

```
create^options := ZSYS^VAL^PCREATOPT^LOWPIN;

error := PROCESS_CREATE_(program^file:pf^length,
                         ! library^file:lf^length  ! ,
                         ! swap^file:sf^length     ! ,
                         ! ext^swapfile:esf^length ! ,
                         ! priority               ! ,
                         ! cpu^number             ! ,
                         process^handle,
                         error^detail,
                         ! name^option:length     ! ,
                         process^desc:max^length,
                         process^desc^length,
                         ! nowait^tag             ! ,
                         ! home^term:home^term^len ! ,
                         ! memory^pages           ! ,
                         ! jobid                  ! ,
                         create^options);
```

Managing a High-PIN Process

This subsection describes how to convert a program to manage a high-PIN process. Managing a process involves these operations:

- ☐ Modifying the state of the process by activating or suspending the process, invoking Inspect or Debug for the process, or stopping or abending the process
- ☐ Getting information about the process
- ☐ Setting process attributes

The following paragraphs describe these operations.

The security restrictions for modifying the state of a process and setting process attributes are described in the *Guardian Procedure Calls Reference Manual*.

Activating a Process

Your existing program might call the ACTIVATEPROCESS procedure to return a process or process pair from the suspended state to the active state:

```
CALL ACTIVATEPROCESS (process^id);
```

Convert the ACTIVATEPROCESS call into a call to the PROCESS_ACTIVATE_ procedure. PROCESS_ACTIVATE_ requires a process handle rather than a process ID as an input parameter to specify the process or process pair to activate.

An optional integer parameter specifies whether only the specified process is activated or both members of a named process pair are activated. Values for this parameter are:

0 (or omitted)	Activate only the process specified by the process handle.
1	If the process specified by the process handle is part of a named process pair, activate both members of the pair.

In the following example, PROCESS_ACTIVATE_ first activates only the process identified by the *process^handle* parameter and then activates both processes of a process pair where one process of the pair is identified by *process^handle*:

```
error := PROCESS_ACTIVATE_(process^handle);

...

! Activate both processes of the process pair.

specifier := activate^pair;   ! Set to 1.

error := PROCESS_ACTIVATE_(process^handle,
                           specifier);  ! Value = 1.
```

Suspending a Process

Your existing program might call the SUSPENDPROCESS procedure to place a process or process pair in the suspended state:

```
CALL SUSPENDPROCESS (process^id);
```

Convert the SUSPENDPROCESS call into a call to the PROCESS_SUSPEND_ procedure. PROCESS_SUSPEND_ requires a process handle rather than a process ID as an input parameter to specify the process or process pair to suspend.

An optional integer parameter specifies whether only the specified process is suspended or both members of a named process pair are suspended. Values for this parameter are:

0 (or omitted)	Suspend only the process specified by the process handle.
1	If the process specified by the process handle is part of a named process pair, suspend both members of the pair.

In the following example, PROCESS_SUSPEND_ first suspends a process identified by the *process^handle* parameter and then suspends both processes of a process pair where one process of the pair is identified by *process^handle*:

```
error := PROCESS_SUSPEND_(process^handle);

...

! Suspend both processes of the process pair.

specifier := suspend^pair;   ! Set to 1.

error := PROCESS_SUSPEND_(process^handle,
                          specifier);  ! Value = 1.
```

Invoking Inspect or Debug for a Process

Your existing program might call the DEBUGPROCESS procedure to invoke Inspect or Debug for another process:

```
CALL DEBUGPROCESS (process^id);
```

Convert the DEBUGPROCESS call to a call to the PROCESS_DEBUG_ procedure. PROCESS_DEBUG_ requires a process handle rather than a process ID as an input parameter to specify the process (for a process other than the calling process).

Optional parameters specify the home terminal and whether the process should enter debug mode immediately. The option to enter debug mode immediately (the debug now option) affects processes where the debug breakpoint is within library code; if the debug now option is set, debugging begins within the library code itself, rather than waiting until the return to user code. The debug now option is available only to the super ID with process access ID (PAID) 255,255.

This example shows two calls to the PROCESS_DEBUG_ procedure:

```
! Debug the process identified by process^handle.

error := PROCESS_DEBUG_(process^handle);

...

! Debug the process identified by process^handle.
! Use the now option (PAID must be 255,255).

now := debug^now;       ! Set to 1.

error := PROCESS_DEBUG_(  process^handle,
                        ! home^terminal ! ,
                          now);     ! Value = 1.
```

Stopping or Abending a Process

Your existing program might call the STOP or ABEND procedure to delete a process or process pair and to send a system message to the creator process indicating that the deletion was caused by a normal (STOP) or abnormal (ABEND) condition:

```
CALL ABEND (process^id,      ! Process to abend.
            ! stop^backup ! ,
            error);

...

CALL STOP;  ! Stop the calling process.
```

Convert the STOP or ABEND call to a call to the PROCESS_STOP_ procedure. PROCESS_STOP_ supersedes both the STOP and ABEND procedures.

Your program can use PROCESS_STOP_ to delete itself, its backup process, or another process. The security restrictions for deleting a process are described in the *Guardian Procedure Calls Reference Manual*.

PROCESS_STOP_ requires a process handle rather than a process ID as an input parameter to specify a process other than the calling process. To stop itself, your program should omit the process-handle parameter or set it to a null value (-1 in each word).

The PROCESS_STOP_ *options*.<15> bit determines whether the system stops or abends a process:

0 Stop the process. The default completion code is 0.

1 Abend the process. The default completion code is 5.

All other bits in the *options* parameter must be zero.

The system sends system message = (Process deletion) to:

- ☐ The mom of the deleted process (if it exists).
- ☐ The ancestor of the deleted process if the deleted process is a single named process or if it is one process of a process pair and both members of the pair are deleted.
- ☐ The job ancestor (GMOM) of the deleted process if the deleted process is part of a batch job.

This example shows PROCESS_STOP_ used to abend a process:

```
! Abend the process identified by process^handle.

options := abend^option;              ! Set to 1.

error := PROCESS_STOP_(process^handle,
                       ! specifier    ! ,
                       options);      ! Value = 1 (abend).
```

These examples show PROCESS_STOP_ used to stop several processes:

```
error := PROCESS_STOP_;  ! Stop the calling process.

...

! Stop the process identified by process^handle.

error := PROCESS_STOP_(process^handle);

...

! Stop the brother process for the calling process.

specifier := stop^brother;                ! Set to 2.

error := PROCESS_STOP_(! process^handle   ! ,
                         specifier);      ! Value = 2.

...

! Stop both processes of a process pair.

specifier := stop^pair;                   ! Set to 1.

error := PROCESS_STOP_(process^handle,
                       specifier);        ! Value = 1.
```

Getting Information About a High-PIN Process

Your existing program might call one of the following C-series procedures to get information about a process. To get information about a high-PIN process, convert your program to call the appropriate D-series procedure.

C-Series Procedure	D-Series Procedure
CREATORACCESSID	PROCESS_GETINFO[LIST]_
GETCPCBINFO	PROCESS_GETINFO[LIST]_
GETCRTPID	PROCESS_GETINFO[LIST]_
GETREMOTECRTPID	PROCESS_GETINFO[LIST]_
LOOKUPPROCESSNAME	PROCESS_GETPAIRINFO_
MOM or MYGMOM	PROCESS_GETINFO[LIST]_
MYTERM	PROCESS_GETINFO[LIST]_
PRIORITY	PROCESS_GETINFO[LIST]_
PROCESSFILESECURITY	PROCESS_GETINFOLIST_
PROCESSINFO	PROCESS_GETINFO[LIST]_
PROCESSTIME	PROCESS_GETINFO[LIST]_
PROGRAMFILENAME	PROCESS_GETINFO[LIST]_

PROCESS_GETINFO_ returns a limited set of information about a specific process identified by its process handle. It also allows a process to retrieve its own process handle.

PROCESS_GETINFOLIST_ returns detailed information about a specific process or about all processes within a CPU that meet a list of search criteria. This procedure also allows you to identify a process using the CPU and PIN if the process handle is not available.

PROCESS_GETPAIRINFO_ returns information about a named process or process pair, including the process handle for the current primary process, the backup process (if one exists), and the ancestor process (if one exists).

Setting Process Attributes for a High-PIN Process

Your existing program might call one of the following C-series procedures to set attributes for a process. To set attributes for a high-PIN process, convert your program to call the appropriate D-series procedure.

C-Series Procedure	D-Series Procedure
ALTERPRIORITY	PROCESS_SETINFO_
PRIORITY	PROCESS_SETINFO_
PROCESSFILESECURITY	PROCESS_SETINFO_
SETMYTERM	PROCESS_SETSTRINGINFO_
STEPMOM	PROCESS_SETINFO_

PROCESS_SETINFO_ alters a single attribute of a process and optionally returns the attribute's previous value. PROCESS_SETINFO_ supersedes PRIORITY, except that it does not return the initial priority. To return the initial priority, use the PROCESS_GETINFOLIST_ procedure.

PROCESS_SETSTRINGINFO_ alters a single string attribute of a process and optionally returns the attribute's previous value.

Both procedures require a process handle as an input parameter to specify the process. A process can specify itself by omitting the process handle input parameter or by setting this parameter to a null value (-1 in each word).

In the following example, PROCESS_SETSTRINGINFO_ sets the home terminal name of the calling process to the value of parameter *new^home^term* and returns the old home terminal name and length in parameter *old^home^term* and *old^home^term^length*. If you use the ZSYSTAL file, set the *home^term^attribute* parameter to ZSYS^VAL^PINF^HOMETERM.

```
home^term^attribute := ZSYS^VAL^PINF^HOMETERM;

error := PROCESS_SETSTRINGINFO_
                (! process^handle     ! ,
                 ! specifier          ! ,
                 home^term^attribute,
                 new^home^term:new^home^term^length,
                 old^home^term:max^length,
                 old^home^term^length);
```

In the following example, PROCESS_SETINFO_ sets the process file security to the value of parameter *new^security* for both members of a named process pair if one of the members of the pair is identified by parameter *process^handle*. If you use the ZSYSTAL file, set the *security^attribute* parameter to ZSYS^VAL^PINF^FILE^SECURITY.

```
security^attribute := ZSYS^VAL^PINF^FILE^SECURITY;

error := PROCESS_SETINFO_(process^handle,
                          specifier,          ! Value = 1.
                          security^attribute,
                          new^security,
                          security^length);
```

For a list of the attributes that you can set using PROCESS_SETINFO_ and PROCESS_SETSTRINGINFO_ , refer to the *Guardian Procedure Calls Reference Manual*.

Opening and Communicating With a High-PIN Server

Your existing program might be a requester that communicates with a server. For example, you might open a server, send it a request, and then process its reply. You might also open a server for a backup requester if your program is running as a process pair.

How much conversion you need to perform depends on whether your server is named or unnamed, and, if the server is named, on how long the name is. Your options are:

- If the server is local and named or if the server is remote with a name of five characters or less (including the dollar sign), then no conversion is necessary. You can still open the high-PIN server using the Guardian C-series-compatible OPEN procedure. See Appendix C, "System Compatibility," for further information on communicating with a named high-PIN process.
- If the server is remote and has a six-character name, then you need to first convert your requester to run at a high PIN as described under "Converting a TAL Program to Run at a High PIN" earlier in this section, and then complete the conversion as described under "Communicating With a High-PIN Server" and "Monitoring a High-PIN Server," later in this section. See Appendix C, "System Compatibility," for further information on communicating with a named high-PIN process.
- If the server is unnamed, then you have the following options:
  - Set the RUNNAMED object-file attribute in the server so that the system provides a name for the server, and pass the system-assigned name to the requester; for example in a DEFINE or an ASSIGN. See "Setting the RUNNAMED Object-File Attribute," later in this section, for details.
  - Convert the requester to run at a high PIN as described under "Converting a TAL Program to Run at a High PIN" earlier in this section, and then complete the conversion as described under "Communicating With a High-PIN Server" and "Monitoring a High-PIN Server," later in this section.

Setting the RUNNAMED Object-File Attribute

The RUNNAMED object-file attribute causes a process to run as a named process even if you do not provide a name for it. Thus, a process can run at a high PIN under the D-series operating system and be opened by an unconverted process using the OPEN procedure.

You set the RUNNAMED object-file attribute either during compilation using a compiler directive or after compilation using the Binder program.

To set the attribute when you compile your program, specify the RUNNAMED directive in your source code or as a compiler option in the TACL RUN command for the TAL compiler. The BINSERV program then sets the RUNNAMED attribute in the object file. An example of this directive (with the HIGHPIN directive) in a source file is:

```
?HIGHPIN, RUNNAMED
```

An example of this directive as a compiler option is:

```
10> TAL /IN talsrc, ... / talobj; HIGHPIN, RUNNAMED
```

You need to specify the RUNNAMED directive only once during a compilation. The TAL compiler ignores redundant RUNNAMED directives.

If you do not set the RUNNAMED attribute when you compile your program, you can set it after compilation using Binder. For a single object file, use the Binder CHANGE command:

```
@CHANGE RUNNAMED ON IN talobj
```

If you are binding more than one object file into a single target object file, use the Binder SET command to set the RUNNAMED object-file attribute. If any of the constituent object files used to build the target file has the RUNNAMED object-file attribute set, Binder sets this attribute in the target object file.

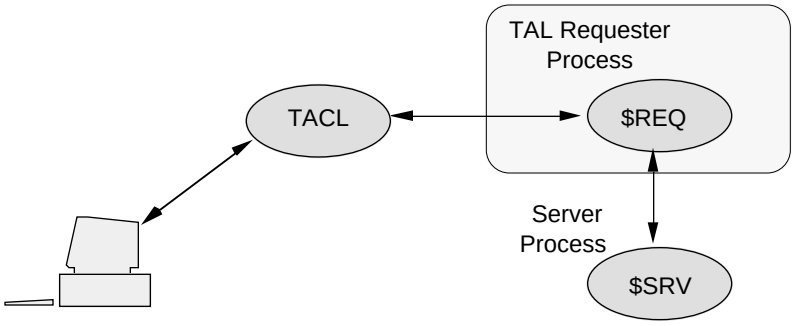
Communicating With a High-PIN Server

A requester can open and communicate with a high-PIN named server by opening the server using the OPEN procedure. However, you must convert your requester to open the server using the FILE_OPEN_ procedure if the server:

- ☐ Is unnamed
- ☐ Is on a remote D-series system and has a six-character name (a dollar sign and five alphanumeric characters)

Figure 3-4 shows the processes involved in converting this part of a typical application. The steps in this subsection apply to the requester process $REQ.

Figure 3-4. Converting a TAL Requester to Communicate With a High-PIN Server



This subsection discusses converting the following operations:

- ☐ Opening and closing the high-PIN server
- ☐ Opening and closing the high-PIN server for a backup process
- ☐ Sending requests to the high-PIN server

Opening a High-PIN Server

Your requester might open the server using the OPEN procedure:

```
INT .server^name[0:11] := ["$SRV", 10 * ["  "]];

...

CALL OPEN (server^name,
           server^file^number,
           nowait^depth,
           sync^depth);
```

Convert your requester to open the high-PIN server using the FILE_OPEN_ procedure. The FILE_OPEN_ procedure requires a variable-length string for the server file-name input parameter rather than the 12-word internal-format file name.

Note If the *file-name* input parameter is incomplete (that is, not fully qualified), FILE_OPEN_ uses the current settings, including the system name, in the =_DEFAULTS DEFINE for the unspecified parts.

FILE_OPEN_ also accepts a DEFINE name that represents a valid file name in this format.

FILE_OPEN_ accepts an integer *options* parameter to specify certain file characteristics. The *options* bit positions represent these options:

options Bit Position	Description
.<0>	Allow unstructured access for a disk file (must be 0 for other files and devices)
.<1>	Execute a nowait open
.<2>	Do not execute an update when the file is opened
.<3>	Use any available file number for backup open (0 means use the same file number as in the primary open)
.<4:13>	Reserved; must be 0
.<14>	Receive C-series system messages ($RECEIVE only)
.<15>	Do not receive process open and close system messages ($RECEIVE only)

The ZSYSTAL file contains LITERAL declarations that you can use with the *options* parameter.

If you started the server using the PROCESS_CREATE_ procedure, you can use the PROCESS_CREATE_ process-descriptor output parameter directly in the FILE_OPEN_ procedure call (shown below as the *server^name* parameter). Refer to "Creating and Managing a High-PIN Process" earlier in this section for details.

```
error := FILE_OPEN_(server^name:server^length,
                    server^file^number,
                    exclusion^mode,
                    nowait^operations,
                    sync^depth,
                    options);
```

If you open the server using the nowait open option, you must call the AWAITIO[X] procedure to complete the open. To determine the *error* and *options* values, call the FILE_GETINFOLIST_ procedure and check the items specified by ZSYS^VAL^FINF^LASTERROR and ZSYS^VAL^FINF^OPENOPTS, respectively (provided you use the ZSYSTAL file).

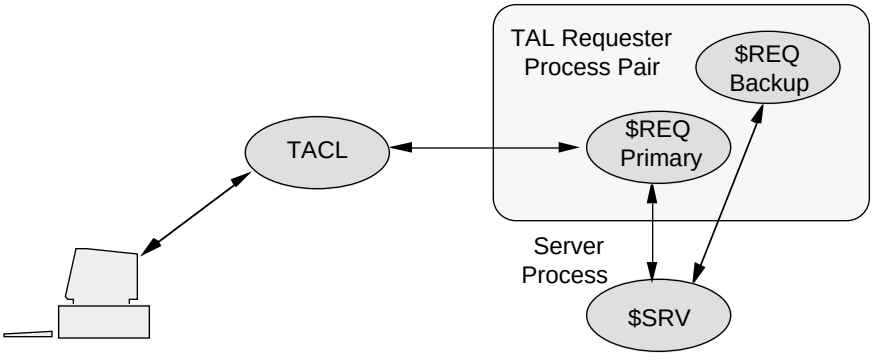
Opening a High-PIN Server for a Backup Requester Process

If your requester is running as a process pair, it might open the server for its backup process using the CHECKOPEN procedure:

```
CALL CHECKOPEN (server^name,
                server^file^number,
                nowait^depth,
                sync^depth,
                ! seq^block^buffer ! ,
                ! buffer^length    ! ,
                back^error);
```

Figure 3-5 shows a requester process pair and a server process.

Figure 3-5. Opening a High-PIN Server for a Backup Process



Convert your requester to open the high-PIN server for its backup process using the FILE_OPEN_CHKPT_ procedure. To identify the high-PIN server, FILE_OPEN_CHKPT_ requires the file number returned by the FILE_OPEN_ procedure call in the primary process. The system returns a file-system error (if a file-system error occurs) in the returned value *error* and the status of the backup open in an output parameter, which is the *backup^open^status* parameter in this example:

```
error := FILE_OPEN_CHKPT_(server^file^number,
                          backup^open^status);
```

If you opened the server using the nowait open option, you must call the AWAITIO[X] procedure to complete the open. To determine the *error* and *backup^open^status* values, call the FILE_GETINFOLIST_ procedure and check the items specified by ZSYS^VAL^FINF^LASTERROR and ZSYS^VAL^FINF^LASTERRORDETAIL, respectively (provided you use the ZSYSTAL file).

Sending a Request to a High-PIN Server

Your requester might send a request to a high-PIN server using the WRITE[X] or WRITEREAD[X] procedure:

```
CALL WRITEREADX (server^file^number,
                 sbuffer,
                 write^count,
                 read^count,
                 count^read);
```

Your WRITE[X] or WRITEREAD[X] procedure call should not require any changes to send a request to a high-PIN server.

Closing a High-PIN Server

Your requester might close the server using the CLOSE procedure:

```
CALL CLOSE (server^file^number);
```

You can close a high-PIN server using either the CLOSE or FILE_CLOSE_ procedure:

```
error := FILE_CLOSE_(server^file^number);
```

Closing a High-PIN Server for a Backup Requester Process

Your requester might close the server for the backup process using the CHECKCLOSE procedure:

```
CALL CHECKCLOSE (server^file^number);
```

You can close the server for the backup process using either the CLOSE procedure or the FILE_CLOSE_CHKPT_ procedure:

```
error := FILE_CLOSE_CHKPT_(server^file^number);
```

Monitoring a High-PIN Server

If your program monitors a high-PIN server, you must convert the following operations:

- ☐ Opening and closing $RECEIVE
- ☐ Reading process-deletion and status-change messages
- ☐ Using the CHILD_LOST_ procedure

The following paragraphs describe how to convert these operations. These steps also can apply to any creator process that monitors a process that it has created.

Opening $RECEIVE

Your requester might open $RECEIVE using the OPEN procedure:

```
INT .receive^name[0:11] := ["$RECEIVE", 8 * ["  "]];

...

CALL OPEN (receive^name,
           receive^file^number,
           read^open^close^msgs,
           receive^depth);
```

Convert your requester to open $RECEIVE using the FILE_OPEN_ procedure. Use a file-name string for the $RECEIVE file name instead of the internal file-name format. Specify the length as a separate integer value.

The *options*.<14> bit must be zero (which is the default value) for the system to send D-series system messages to $RECEIVE; otherwise, the system sends C-series system messages to $RECEIVE for the requester.

An example of a FILE_OPEN_ procedure call for $RECEIVE is:

```
LITERAL receive^name^length = 8;

STRING .receive^name[0:receive^name^len-1] := ["$RECEIVE"];

...

! Open $RECEIVE to read D-series system messages.

error := FILE_OPEN_(receive^name:receive^name^length,
                    receive^file^number,
                    ! access^mode       ! ,
                    ! exclusion^mode    ! ,
                    ! nowait^operations ! ,
                    receive^depth,
                    options);
```

Reading System Messages From $RECEIVE

Your requester might read system messages from $RECEIVE using the READ[X] or READUPDATE[X] procedure:

```
STRING .message^buffer[0:199];  ! Message buffer (200 bytes).

...

read^count := 200;

CALL READX (receive^file^number,
            message^buffer,
            read^count,
            bytes^read);
```

The lengths shown for each system message are subject to change. In a future release, Tandem might add new fields to the end of a system message (while maintaining the layout of the existing fields). Therefore, use a READ[X] or READUPDATE[X] message buffer at least 250 bytes in length. Also, use a *read^count* parameter of 250 bytes.

If you use the ZSYSTAL file, use the ZSYS^VAL^SMSG^LEN LITERAL declaration to specify the system message length in bytes. If you work in words you can use the ZSYS^VAL^SMSG^WLEN LITERAL declaration instead.

```
STRING .message^buffer[0:ZSYS^VAL^SMSG^LEN - 1];

...

read^count := ZSYS^VAL^SMSG^LEN;

CALL READX (receive^file^number,
            message^buffer,
            read^count,
            bytes^read);
```

The ZSYSTAL file also contains structures that you can use when your requester reads system messages.

Reading Process-Deletion System Messages. Your requester might monitor a server process by reading these process-deletion system messages from $RECEIVE:

-2 CPU down: named process deletion
-5 Process normal deletion: stop
-6 Process abnormal deletion: abend

Convert your requester to read and process the D-series system message -101 (Process deletion), which supersedes all the above messages.

Reading Status-Change System Messages. Your requester might monitor a server process by reading these status-change system messages from $RECEIVE:

-2 CPU down: local CPU failure after process called MONITORCPUS
-8 Change in status of network node

Continue to read system message -2. Then, convert your requester to read these new status-change messages, all of which supersede system message -8:

-100 Remote CPU down
-110 Loss of communication with node
-113 Remote CPU up

To receive system messages -100, -110, and -113, first call the MONITORNET procedure with the *enable* parameter set to 1.

Processing System Messages Using the CHILD_LOST_ Procedure

Your requester might call a user-written routine to determine whether a process-deletion or status-change message affects the server.

You might convert your requester to call the new CHILD_LOST_ procedure. The CHILD_LOST_ procedure accepts the process handle of a process you are monitoring and either a C-series (-2, -5, -6, or -8) or D-series (-2, -100, -101, -110, or -113) process-deletion or status-change system message:

```
error := CHILD_LOST_(message:message^length,
                     process^handle);
```

The CHILD_LOST_ *error* returned value indicates whether the process (or process pair) is lost:

0 The process (or process pair) is not lost.
4 The process (or process pair) is lost.

Note System message -101 (Process deletion) contains the process handle and process descriptor of the process that terminated. If a named process (or process pair) has terminated, this is the last opportunity for you to save the process name of the process (or process pair).

Closing $RECEIVE

Your requester might close $RECEIVE using the CLOSE procedure:

```
CALL CLOSE (receive^file^number);
```

You can close $RECEIVE using either the CLOSE or FILE_CLOSE_ procedure:

```
error := FILE_CLOSE_(receive^file^number);
```

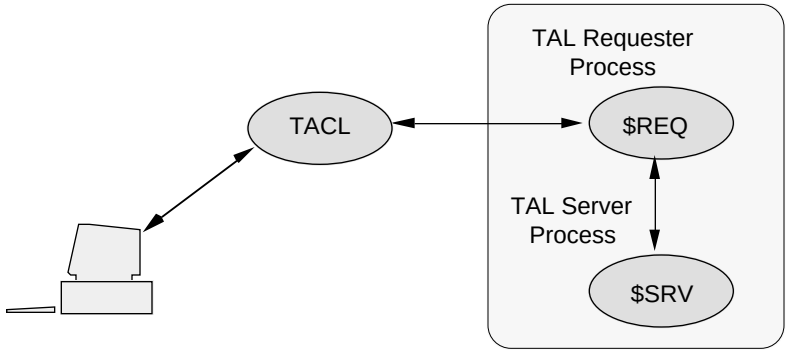
Allowing a High-PIN Creator

This subsection describes two approaches to allowing your process to be created by a high-PIN process:

- ☐ Convert your program to use the D-series enhanced interface when obtaining the process identifier of the creator process or when receiving the startup sequence of messages if your process does not use the INITIALIZER procedure.
- ☐ Set the HIGHREQUESTERS object file attribute on your program file.

Figure 3-6 shows the sample application. The box shows which processes this part of the conversion applies to. Allowing a high-PIN creator applies to both of these processes.

Figure 3-6. Converting a TAL Program to Allow a High-PIN Creator



Full Conversion or HIGHREQUESTERS?

Tandem recommends conversion to the D-series enhanced interface—this method works in all cases. However, in many cases, the HIGHREQUESTERS attribute provides an acceptable short cut.

You should not use the HIGHREQUESTERS method if either of the following is true:

- ☐ Your process cannot allow high-PIN openers other than the process creator. Using the HIGHREQUESTERS flag allows any high-PIN opener for the life of the process.
- ☐ The process ID of the creator process requires further processing for which a synthetic process ID is inadequate; for example, when you compare it with another process identifier or pass it to another process, log, or operator.

If You Do a Full Conversion

To use the D-series enhanced interface to allow your program to enable a high-PIN creator, you need to convert your program to run at a high PIN as described in "Converting a TAL Program to Run at a High PIN," earlier in this section. In addition, you might choose to make changes if your program obtains the identity of its creator or if your program does not use the INITIALIZER procedure to process the startup sequence of messages.

If your program directly obtains the identity of its creator, you should:

- ☐ Convert all calls to the MOM procedure into calls to the PROCESS_GETINFO_ procedure.
- ☐ Convert all calls to the LOOKUPPROCESSNAME procedure that return the identity of the creator into calls to the PROCESS_GETPAIRINFO_ procedure.

PROCESS_GETINFO_ and PROCESS_GETPAIRINFO_ return process handles instead of the process IDs that the MOM and LOOKUPPROCESSNAME procedures return. Process IDs are unsuitable because the PIN field is too short to return a high PIN; you get a synthetic process ID instead, which always has a PIN of 255. "Getting Your Creator's Process Identifier," later in this section, provides details.

If your program does not use the INITALIZER procedure, you must:

- ☐ Open $RECEIVE using the FILE_OPEN_ procedure instead of the OPEN procedure and then read and process the D-series messages.
- ☐ Convert calls to the RECEIVEINFO procedure and the LASTRECEIVE procedure into calls to the FILE_GETRECEIVEINFO_ procedure.

"Converting a Startup Sequence That Does Not Use INITIALIZER," later in this section, provides details.

If You Use the HIGHREQUESTERS Attribute

If you choose to set the HIGHREQUESTERS object-file attribute, then the high-PIN creator of your process is allowed to open your process. "Setting the HIGHREQUESTERS Attribute to Allow a High-PIN Creator," later in this section, provides details on how to do this.

Further conversion is unnecessary if synthetic process IDs returned by the MOM, LOOKUPPROCESSNAME, RECEIVEINFO, and LASTRECEIVE procedures are enough to distinguish the creator process for the needs of your application.

Getting Your Creator's Process Identifier

Your existing application might get the process ID of its creator directly using either of:

- ☐ The MOM procedure
- ☐ The LOOKUPPROCESSNAME procedure

In either case, you must convert your program to use the corresponding D-series procedure call as described in the following paragraphs.

Converting MOM Procedure Calls

Your existing application might get the process ID of its creator using the MOM procedure as follows:

```
INT moms^process^id[0:3];

CALL MOM(moms^process^id);
```

Convert your program to use the D-series PROCESS_GETINFO_ procedure to return the process handle instead:

```
INT my^process^handle[0:ZSYS^VAL^PHANDLE^WLEN - 1];
INT moms^process^handle[0:ZSYS^VAL^PHANDLE^WLEN - 1];

my^process^handle ':=' ZSYS^VAL^PHANDLE^WLEN * [-1];
CALL PROCESS_GETINFO_(my^process^handle,
                      !process^descriptor!,
                      !process^descriptor^length!,
                      !priority!,
                      moms^process^handle);
```

Converting LOOKUPPROCESSNAME Procedure Calls

Your existing application might get the process ID of its creator using the LOOKUPPROCESSNAME procedure as follows:

```
INT ppd[0:9];
INT momsid[0:3];

ppd ':=' "$MYMOM                  ";
CALL LOOKUPPROCESSNAME(ppd);
momsid ':=' ppd[5] FOR 4;
```

Convert your program to use the D-series PROCESS_GETPAIRINFO_ procedure to return the process handle instead. This example returns the process handle of the ancestor process:

```
INT ancestor^process^handle[0:ZSYS^VAL^PHANDLE^WLEN - 1];

CALL PROCESS_GETPAIRINFO_(!process^handle!,
                          !process^name:max^len!,
                          !pair^length!,
                          !primary^process^handle!,
                          !backup^process^handle!,
                          !search index!,
                          ancestor^process^handle);
```

Converting a Startup Sequence That Does Not Use INITIALIZER

This subsection describes the remaining steps for conversion assuming that you have already converted any attempts to directly obtain the identity of your creator process and decided that using the HIGHREQUESTERS attribute is inappropriate.

For a program that does not use INITIALIZER, you need to convert the way you process the startup sequence as follows:

- ☐ Use the FILE_OPEN_ procedure instead of OPEN when opening $RECEIVE to read the startup sequence, and ask for D-series system messages.
- ☐ Convert your code to handle D-series system messages, specifically system messages -103 (Process open) and -104 (Process close).
- ☐ Use the FILE_GETRECEIVEINFO_ procedure instead of RECEIVEINFO or LASTRECEIVE if your program uses either of these procedures to identify the sender of the startup sequence.

Opening $RECEIVE to Read the Startup Sequence

If your new process does not use INITIALIZER to process the startup sequence, then it typically opens $RECEIVE using the OPEN procedure with the OPEN *flags*.<1> bit set to 1 (*flags* = %40000). This allows you to receive system messages such as -30 (Process open) and -31 (Process close). Your existing program might open $RECEIVE as follows:

```
INT .receive^name[0:11] := ["$RECEIVE", 8 * ["  "]];

LITERAL read^open^close^msgs = %40000 ;

...

CALL OPEN (receive^name,
           receive^file^number,
           read^open^close^msgs,   ! Value = %40000
           receive^depth);
```

Convert your program to open $RECEIVE for processing the startup sequence using the FILE_OPEN_ procedure:

1. Use a file-name string for the $RECEIVE file name instead of the internal file-name format. Specify the length as a separate integer value.
2. Make sure that the FILE_OPEN_ *options*.<15> bit is zero (the default value). If this bit is not zero, system messages such as -103 (Process open) and -104 (Process close) are not sent to $RECEIVE.
3. Make sure that the FILE_OPEN_ *options*.<14> bit is zero (the default value) so that the system sends D-series system messages to $RECEIVE. If this bit is not zero, the system sends C-series system messages to $RECEIVE.
4. Set any other FILE_OPEN_ input parameters as required and call the procedure:

```
LITERAL receive^name^length = 8;

STRING .receive^name[0:receive^name^len-1] :=
                                    ["$RECEIVE"];

...

! Open $RECEIVE to read D-series system messages.

error := FILE_OPEN_(receive^name:receive^name^length,
                    receive^file^number,
                    ! access^mode        ! ,
                    ! exclusion^mode     ! ,
                    nowait^operations,
                    receive^depth);
```

If you open $RECEIVE using the FILE_OPEN_ procedure, the system assumes that you support high-PIN requesters (provided the *options*.<14> bit is zero). You do not need to explicitly set the HIGHREQUESTERS object-file attribute in your server's object file.

When you close $RECEIVE, use either the CLOSE or FILE_CLOSE_ procedure.

Reading and Processing Process Open and Process Close System Messages

When processing the startup sequence, your program might read the C-series -30 (Process open) and -31 (Process close) system messages from $RECEIVE.

To allow a high-PIN creator process, convert your server to read the D-series -103 (Process open) and -104 (Process close) system messages.

To process D-series system messages correctly, you need to change the size of the read buffer to allow for longer D-series system messages. Your existing process might read these messages using the READ[X] or READUPDATE[X] procedure in code similar to the following:

```
STRING .message^buffer[0:199];  ! Message buffer (200 bytes).

...

read^count := 200;

CALL READX (receive^file^number,
            message^buffer,
            read^count,
            bytes^read);
```

The lengths shown for each system message are subject to change. Use a READ[X] or READUPDATE[X] message buffer at least 250 bytes in length. Also, use a *read^count* parameter value of 250 bytes.

If you use the declarations in the ZSYSTAL file, use the ZSYS^VAL^SMSG^LEN LITERAL for the system-message length in bytes or the ZSYS^VAL^SMSG^WLEN LITERAL for the length in words:

```
STRING .message^buffer[0:ZSYS^VAL^SMSG^LEN - 1];  ! Message
                                                  !  buffer (250 bytes)
...

read^count := ZSYS^VAL^SMSG^LEN;

CALL READX (receive^file^number,
            message^buffer,
            read^count,
            bytes^read);
```

Getting Information About the Process Open Message in the Startup Sequence

The RECEIVEINFO and LASTRECEIVE procedures obtain information about the last message read from $RECEIVE. Your existing program might call one of these procedures after reading the process-open message to obtain the identity of the process that sent the message, that is, the process creator:

```
CALL RECEIVEINFO (process^id,
                  message^tag,
                  sync^id,
                  file^number,
                  read^count,
                  io^type);
```

Convert the RECEIVEINFO or LASTRECEIVE call into a call to the FILE_GETRECEIVEINFO_ procedure:

```
! Return information about the last message.

error := FILE_GETRECEIVEINFO_(message^info);
creators^process^handle ':=' message^info[6] FOR
                             ZSYS^VAL^PHANDLE^WLEN;
```

FILE_GETRECEIVEINFO_ returns information in the 17-word *message^info* parameter. The process handle of the creator is in words 6 through 15.

See Table 3-1, later in this section, for a summary of all the information returned in the *message^info* parameter.

Setting the HIGHREQUESTERS Attribute to Allow a High-PIN Creator

You can set the HIGHREQUESTERS object-file attribute in your source file, or you can set it after you have finished converting your source code, either during compilation using a compiler directive or after compilation using the Binder program. The following paragraphs describe all of these alternatives.

Setting the HIGHREQUESTERS Compiler Directive

To set the attribute when you compile your program, specify the HIGHREQUESTERS compiler directive in your source code or as a compiler option in the TACL RUN command for the TAL compiler. The BINSERV program then sets the HIGHREQUESTERS attribute in the object file. An example of this directive in a source file is:

```
?HIGHREQUESTERS
```

An example of this directive as a compiler option is:

```
10> TAL /IN talsrc, ... / talobj; HIGHREQUESTERS
```

You need to specify the HIGHREQUESTERS directive only once during a compilation. If your program file copies source code from another file, specify the HIGHREQUESTERS directive only in the program file that contains the main procedure; do not specify the directive in the other file (or files).

Setting the HIGHREQUESTERS Attribute Using Binder

If you do not set the HIGHREQUESTERS attribute when you compile your program, you can set it after compilation using Binder. For a single object file, use the Binder CHANGE command:

```
@CHANGE HIGHREQUESTERS ON IN TALOBJ
```

If you are binding more than one object file into a single target object file, use the Binder SET command to set the HIGHREQUESTERS object-file attribute. For Binder to set the HIGHREQUESTERS object-file attribute in a target object file, the object file containing the main procedure must have this object-file attribute set.

For more information about the HIGHREQUESTERS object-file attribute, refer to “Allowing Opens by High-PIN Requesters” in Appendix C, “System Compatibility.”

Being Opened by and Communicating With a High-PIN Requester

This subsection describes how to convert a TAL server to communicate with a high-PIN requester. Whether you need to convert the server process depends in part on whether the server tracks its openers. If the server does keep track of its openers, you should enable the server to run at a high PIN as described in "Converting a TAL Program to Run at a High PIN," earlier in this section, and then convert the server as described under "Converting a Server," later in this subsection.

If the server does not track its openers, or if you choose not to perform the conversion, then you can keep the server process at a low PIN and not convert it, except for setting the HIGHREQUESTERS object-file attribute as described under "Setting the HIGHREQUESTERS Attribute to Allow High-PIN Openers," later in this subsection. Setting this attribute enables a high-PIN requester to open a low-PIN server.

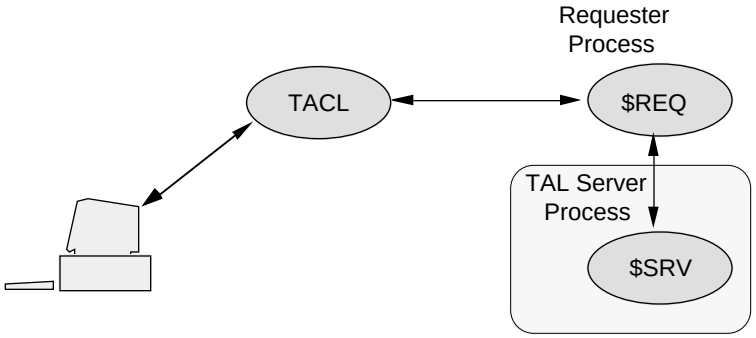
Converting a Server

If your server process tracks its openers, you must convert the following parts of your program:

- ☐ Defining an opener table
- ☐ Opening $RECEIVE
- ☐ Reading D-series system messages from $RECEIVE
- ☐ Getting information about system messages
- ☐ Processing system messages
- ☐ Replying to a system message
- ☐ Using the OPENER_LOST_ procedure to maintain an opener table

Figure 3-7 shows the processes involved in converting an application. The steps described in this subsection apply to the server process $SRV.

Figure 3-7. Converting a TAL Server to Communicate With a High-PIN Requester



Defining an Opener Table

If your server tracks its openers, it might define an opener table that uses a process ID to identify an opener (primary process opener and backup process opener):

```
STRUCT .opener^table;
  BEGIN
    INT current^count;
    STRUCT openers [0:max^openers - 1];
      BEGIN
        INT primary^process^id [0:3];
        INT primary^file^number;
        INT backup^process^id  [0:3];
        INT backup^file^number;
      END;
  END;
```

Convert your opener table to identify an opener using a process handle rather than a process ID. To use the OPENER_LOST_ procedure (which is described later in this subsection) to manage your opener table, define the table as follows:

- Use a process handle to identify both a primary-process and backup-process opener.
- Declare the process-handle field for the backup-process opener immediately after the process-handle field for the primary-process opener (that is, the fields must be stored in a 20-word contiguous part of an entry).
- Declare table entries as fixed length and contiguous.
- Do not store variable-length items in the table. If necessary, save a pointer in the table to a variable-length item.
- Set the process handles for primary and backup openers in unused entries to null values (all -1s).

An example of an opener table that the OPENER_LOST_ procedure can process is:

```
STRUCT .opener^table;
  BEGIN
    INT current^count;
    STRUCT openers [0:max^openers - 1];
      BEGIN
        INT primary^process^handle
            [0:ZSYS^VAL^PHANDLE^WLEN - 1];
        INT backup^process^handle
            [0:ZSYS^VAL^PHANDLE^WLEN - 1];
        INT primary^file^number;
        INT backup^file^number;
      END;
  END;
```

Opening $RECEIVE

Your server might open $RECEIVE using the OPEN procedure with the OPEN *flags*.<1> bit set to 1 (*flags* = %40000). This allows you to receive system messages such as -30 (Process open) and -31 (Process close):

```
INT .receive^name[0:11] := ["$RECEIVE", 8 * ["  "]];

LITERAL read^open^close^msgs = %40000 ;

...

CALL OPEN (receive^name,
           receive^file^number,
           read^open^close^msgs,   ! Value = %40000.
           receive^depth);
```

Convert your server to open $RECEIVE using the FILE_OPEN_ procedure:

1. Use a file-name string for the $RECEIVE file name instead of the internal file-name format. Specify the length as a separate integer value.
2. Make sure that the FILE_OPEN_ *options*.<15> bit is zero (the default value). If this bit is not zero, system messages such as -103 (Process open) and -104 (Process close) are not sent to $RECEIVE.
3. Make sure that the FILE_OPEN_ *options*.<14> bit is zero (the default value) so that the system sends D-series system messages to $RECEIVE. If this bit is not zero, the system sends C-series system messages to $RECEIVE.
4. Set any other FILE_OPEN_ input parameters as required and call the procedure:

```
LITERAL receive^name^length = 8;

STRING .receive^name[0:receive^name^length-1] :=
                                    ["$RECEIVE"];

...

! Open $RECEIVE to read D-series system messages.

error := FILE_OPEN_(receive^name:receive^name^length,
                    receive^file^number,
                    ! access^mode        ! ,
                    ! exclusion^mode     ! ,
                    nowait^operations,
                    receive^depth);
```

If you open $RECEIVE using the FILE_OPEN_ procedure, the system assumes that you support high-PIN requesters (provided the *options*.<14> bit is zero). You do not need to explicitly set the HIGHREQUESTERS object-file attribute in your server's object file.

When you close $RECEIVE, use either the CLOSE or FILE_CLOSE_ procedure.

Reading System Messages From $RECEIVE

Your existing server might read system messages from $RECEIVE using the READ[X] or READUPDATE[X] procedure using code similar to the following:

```
STRING .message^buffer[0:199];  ! Message buffer (200 bytes).

...

read^count := 200;

CALL READX (receive^file^number,
            message^buffer,
            read^count,
            bytes^read);
```

The lengths shown for each system message are subject to change. Use a READ[X] or READUPDATE[X] message buffer at least 250 bytes in length. Also, use a *read^count* parameter value of 250 bytes.

If you use the declarations in the ZSYSTAL file, use the ZSYS^VAL^SMSG^LEN LITERAL for the system-message length in bytes or the ZSYS^VAL^SMSG^WLEN LITERAL for the length in words:

```
STRING .message^buffer[0:ZSYS^VAL^SMSG^LEN - 1];  ! Message
                                                  !  buffer (250 bytes)
...

read^count := ZSYS^VAL^SMSG^LEN;

CALL READX (receive^file^number,
            message^buffer,
            read^count,
            bytes^read);
```

Getting Information About System Messages

Your server might call the RECEIVEINFO or LASTRECEIVE procedure to obtain information about the last message read from $RECEIVE:

```
CALL RECEIVEINFO (process^id,
                  message^tag,
                  sync^id,
                  file^number,
                  read^count,
                  io^type);
```

Convert the RECEIVEINFO or LASTRECEIVE call into a call to the FILE_GETRECEIVEINFO_ procedure:

```
! Return information about the last message.

error := FILE_GETRECEIVEINFO_(message^info);
```

FILE_GETRECEIVEINFO_ returns information in the 17-word *message^info* parameter, which has the format shown in Table 3-1. The ZSYSTAL file contains a structure that you can use for the *message^info* format.

Table 3-1. FILE_GETRECEIVEINFO_ message^info Parameter Format

Word	Description
0	I/O type for the message: 0 = A system message was sent. 1 = The sender called WRITE[X]. 2 = The sender called READ[X]. 3 = The sender called WRITEREAD[X].
1	The maximum reply count in bytes
2	The message tag identifying the message
3	The file number for the message
4 through 5	The sync ID for the message
6 through 15	The process handle of the process sending the message
16	The *open^label* from a previous reply (or -1 if unavailable or for a C-series message)

Reading and Processing Open and Close System Messages

To monitor an opener, your server might read the C-series -30 (Process open) and -31 (Process close) system messages from $RECEIVE.

To monitor a high-PIN process, convert your server to read the D-series -103 (Process open) and -104 (Process close) system messages. When your server is opened or closed by a process pair, it receives a process-open or a process-close message from each process of the pair.

If you call the RECEIVEINFO or LASTRECEIVE procedure to obtain information about the process-open or process-close message, convert the call into a call to the FILE_GETRECEIVEINFO_ procedure as described under “Getting Information About System Messages,” earlier in this section.

After calling FILE_GETRECEIVEINFO_, update your opener table using the process handle rather than the process ID to identify the opener.

System Message -103 (Process Open). Check the process open *sysmsg*[7].<15> bit (or ZSYS^DDL^SMSG^OPEN.Z^FLAGS if you use the ZSYSTAL file), which indicates whether the opener is a primary or backup process:

- ☐ Primary open (*sysmsg*[7].<15> bit = 0): Add an entry in your opener table for the process.
- ☐ Backup open (*sysmsg*[7].<15> bit = 1): Process a backup open as follows:
  1. Get the process handle for the primary opener from the process-open system message (-103). This process handle is in *sysmsg*[8] for ten words (or the ZSYS^DDL^SMSG^OPEN.Z^PRIMARY field if you use the ZSYSTAL file).
  2. Use the process handle to search your opener table for the corresponding primary-process open entry. If you find this entry but there is no backup open yet (the backup process handle is null), add the backup process handle to the table entry.
  3. If the primary-process open entry is not found, reject the backup open with a file-system error greater than 9.

System Message -104 (Process Close). Delete the opener-table entry for this process. You should receive a process-close message from each process of a process pair.

Reading and Processing Status-Change Messages

If one of your openers has a CPU failure, or if its system fails or becomes partitioned from your system because of a network failure, you do not receive a process-close message (-31). Therefore, when maintaining an opener table, your server might read and process these status-change messages:

-2	CPU down: local CPU failure after the process called MONITORCPUS
-8	Change in status of network node

Continue to read system message -2. In addition, read these new status-change messages (all of which supersede C-series system message -8):

-100	Remote CPU down
-110	Loss of communication with node
-113	Remote CPU up

To receive system messages -100, -110, and -113, first call the MONITORNET procedure with the *enable* parameter set to 1.

Replying to a System Message

Your server might reply to a system message using the Guardian REPLY[X] procedure:

```
CALL REPLYX (reply^buffer,
             write^count,
             count^written,
             message^tag,
             error^return);
```

Replying to System Message -103 (Process Open). The D-series system supports returning a label value in the reply to a system message -103 (Process open). Typically, an opener-table index gets sent in this way. This label then appears in the *open^label* field of future FILE_GETRECEIVEINFO_ procedure calls that provide information about messages received from the same requester. To support this feature, the file system expects a reply buffer with a length of 0 to 4 bytes; otherwise, the open in the requester returns an error.

Your server might reply to an open message as follows:

```
write^count := any^valid^integer;
CALL REPLYX(reply^buffer,
            write^count,
            ! count^written ! ,
            ! message^tag   ! ,
            error^return);
```

To make use of the *open^label* field in the FILE_GETRECEIVEINFO_ procedure, you must convert your code to reply to the open message as follows:

```
reply^buffer[0] := -103;
reply^buffer[1] := open^label^value;
write^count := 4;
CALL REPLYX (reply^buffer,
             write^count,
             ! count^written ! ,
             ! message^tag   ! ,
             error^return);
```

If you do not want to use the *open^label* field, you still need to be sure that the reply buffer has a length of 0 to 4 bytes. Convert your server as follows:

```
write^count := 0;
CALL REPLYX (reply^buffer,
             write^count,
             ! count^written ! ,
             ! message^tag   ! ,
             error^return);
```

Replying to an Unknown System Message. Your server should be able to handle an unknown system message. If the first word of a message contains an unknown message number, call the REPLY[X] procedure with an error indication of 2 (invalid operation):

```
CALL REPLYX (! reply^buffer  ! ,
             ! write^count   ! ,
             ! count^written ! ,
             ! message^tag   ! ,
             invalid^operation);   ! Value = 2.
```

Using the OPENER_LOST_ Procedure to Maintain an Opener Table

After receiving a status-change message, your server might call one or more routines to maintain its opener table.

You might want to use the OPENER_LOST_ procedure to maintain your opener table. OPENER_LOST_ determines whether a status-change message affects your opener table and updates the appropriate table entry if an opener was lost.

OPENER_LOST_ accepts a C-series (-2 or -8) or D-series (-2, -100, -110, or -113) status-change message and searches your opener table for any processes affected by the message. If OPENER_LOST_ determines that an opener has been lost, it updates the opener-table entry and returns the index of the entry and an *error* value. The *error* value indicates the reason for the opener-table change:

error Value	Reason
4	A backup process opener is lost
5	A primary process opener is lost; the backup process is now the primary process
6	The primary process and backup process (if it exists) openers for a table entry are lost; the table entry is now free

When OPENER_LOST_ returns an *error* value of zero, processing is complete for the message.

To process all entries in your opener table for a status-change message, set up a loop similar to the one shown below. The opener table must be defined as described under "Defining an Opener Table," earlier in this subsection.

```
done  :=  0;    ! Set control for start of loop.
index := -1;    ! Set index for start of loop.

DO BEGIN
  error := OPENER_LOST_(message:message^length,
                        opener^table.openers,
                        index,
                        opener^table.current^count,
                        $LEN(opener^table.openers));

  CASE error OF
  BEGIN
     4 ->          ! Processing for lost backup opener
     5 ->          ! Processing for lost primary opener
     6 ->          ! Processing for lost opener
                   !  (primary and backup for a process pair)
     OTHERWISE -> done = -1   ! Processing is finished or
                              !  error occurred
  END;
END
UNTIL done;
```

Setting the HIGHREQUESTERS Attribute to Allow High-PIN Openers

The HIGHREQUESTERS object-file attribute allows a process to support requests from high-PIN requesters. You can set the HIGHREQUESTERS attribute by including a compiler directive in your source file, or you can set it after you have finished converting your source code either, with a compiler option or after compilation using the Binder program.

To set the attribute when you compile your program, specify the HIGHREQUESTERS compiler directive in your source code or as a compiler option in the TACL RUN command for the TAL compiler. The BINSERV program then sets the HIGHREQUESTERS attribute in the object file. An example of this directive in a source file is:

```
?HIGHREQUESTERS
```

An example of this directive as a compiler option is:

```
10> TAL /IN talsrc, ... / talobj; HIGHREQUESTERS
```

You need to specify the HIGHREQUESTERS directive only once during a compilation. If your program file copies source code from another file, specify the HIGHREQUESTERS directive only in the program file that contains the main procedure; do not specify the directive in the other file (or files).

If you do not set the HIGHREQUESTERS attribute when you compile your program, you can set it after compilation using Binder. For a single object file, use the Binder CHANGE command:

```
@CHANGE HIGHREQUESTERS ON IN TALOBJ
```

If you are binding more than one object file into a single target object file, use the Binder SET command to set the HIGHREQUESTERS object-file attribute. For Binder to set the HIGHREQUESTERS object-file attribute in a target object file, the object file containing the main procedure must have this object-file attribute set.

For more information about the HIGHREQUESTERS object-file attribute, refer to "Allowing Opens by High-PIN Requesters" in Appendix C, "System Compatibility."

4 Converting COBOL85 Applications

A COBOL85 program can run at a low PIN under the D-series operating system without any changes. However, for a program to use the extended features of the D-series operating system, specific parts of it must be converted. The topics in this section are:

- ☐ Converting basic elements of a COBOL85 program, such as using the ZSYSCOB file, declaring variables, calling Guardian procedures, and running the COBOL85 compiler
- ☐ Converting a COBOL85 program to run at a high PIN
- ☐ Converting a COBOL85 program to create and manage a high-PIN process
- ☐ Converting a requester to communicate with a high-PIN server
- ☐ Converting a server to communicate with a high-PIN requester

Section 8, “Converting Other Parts of an Application,” contains information about converting other parts of a COBOL85 application. For additional information about the Tandem implementation of COBOL85, refer to the *COBOL85 Reference Manual*.

Converting Basic Elements of a COBOL85 Program

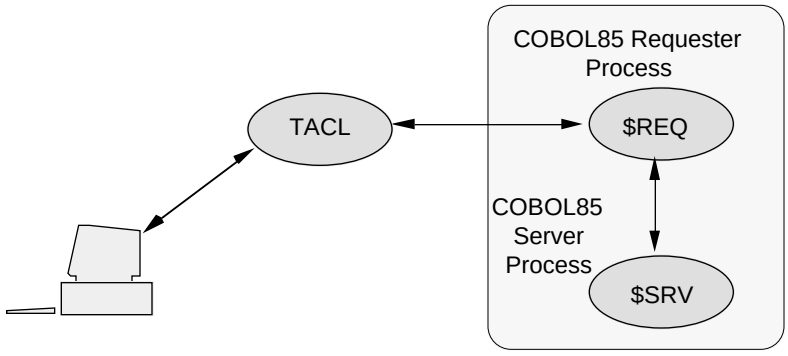
This subsection describes conversion that applies to all COBOL85 programs you need to convert to run under the D-series operating system, irrespective of what the program does. Later subsections describe how to convert specific functions of your programs, such as communicating with a high-PIN process.

This subsection discusses the following topics:

- ☐ Using source declarations from the ZSYSCOB file
- ☐ Declaring and using variables for high PINs, file-system error numbers, file names, and process identifiers
- ☐ Calling Guardian procedures
- ☐ Converting for new reserved words
- ☐ Running the COBOL85 compiler
- ☐ Binding Converted Object Files

Figure 4-1 shows a typical application. The box shows which processes this part of the conversion applies to. Converting basic elements of a COBOL85 program applies to both of these processes.

Figure 4-1. Converting Basic Elements of a COBOL85 Program



Using the ZSYSCOB Declarations

Tandem provides COBOL85 source declarations of data items and structures for Guardian procedures and system messages in the ZSYSCOB file. This file is typically found on the $SYSTEM.ZSYSDEFS subvolume. Contact your system manager to find the location of this file on your system.

To use the declarations in this file, include them in your source code using the COPY statement. For example, this statement copies the entire ZSYSCOB file:

```
COPY "$SYSTEM.ZSYSDEFS.ZSYSCOB".
```

The ZSYSCOB file is divided into sections, which allows you to copy only the sections your program actually needs. For example, these statements copy only the process creation and system-message constant declarations:

```
COPY PROCESS-CONSTANT         OF "$SYSTEM.ZSYSDEFS.ZSYSCOB".
COPY SYSTEM-MESSAGES-CONSTANT OF "$SYSTEM.ZSYSDEFS.ZSYSCOB".
```

To print a listing of the ZSYSCOB file to check the declarations that are available for your program, use the FUP COPY command:

```
10> FUP COPY $SYSTEM.ZSYSDEFS.ZSYSCOB, $s.#lineptr
```

Declaring and Using Programming Variables

For your existing program to run at a high PIN, you might need to add or modify declarations for the following variables:

- ☐ CPU and PIN data items
- ☐ File-system error numbers
- ☐ Guardian file names, including disk file names, device names, and process file names
- ☐ Process identifiers including process IDs, process handles, and process descriptors

These declarations are described in the following paragraphs.

Declaring CPU and PIN Data Items

When using Guardian procedures that return values for CPU number and PIN, your existing program might declare either a two-digit data item for the CPU value and a three-digit data item for the PIN value, or a just a three-digit data item for a PIN value:

```
WORKING-STORAGE SECTION.
01 CPU-PIN-DEFINITIONS.
   05 CPU             PIC S9(2) COMPUTATIONAL.
   05 PIN             PIC S9(3) COMPUTATIONAL.
```

Use a USAGE IS NATIVE-2 clause in the declaration of all PIN values, including PINs for backup processes, to allow up to 32,767 PINs. Declare a CPU number as a separate two-digit unsigned data item:

```
WORKING-STORAGE SECTION.
01 CPU-PIN-DEFINITIONS.
   05 CPU             PIC S9(2) COMPUTATIONAL.
   05 PIN             NATIVE-2.
```

Note that you can declare each PIN using a PIC 9(4) COMPUTATIONAL clause if you will never need a PIN higher than 9,999.

Declaring and Checking File-System Error Numbers

A Guardian procedure can return a file-system error number in the GIVING phrase of an ENTER statement to report an error or special condition. Similarly, the COBOLFILEINFO utility routine is capable of returning a file-system error code. You might need to convert the parts of your program that declare and process file-system errors.

For example, your program might declare a three-digit data item for a file-system error number:

```
* C-series file-system error number.

 01  ERROR-NUMBER     PIC S9(3) COMPUTATIONAL.
```

Declare file-system error numbers as five-digit data items:

```
* D-series file-system error number.

 01  ERROR-NUMBER     PIC S9(5) COMPUTATIONAL.
```

You can also declare a data item for a file-system error number using the USAGE IS NATIVE-2 clause.

Your program might include a routine that assumes a maximum value for a file-system error number (for example, 255). A D-series file-system error number is a maximum of 16 bits. Therefore, make sure that your routine does not exclude any new error numbers. Also, because Tandem might define additional error numbers in future releases, do not consider currently undefined numbers as invalid.

For a list and description of all file-system error numbers, refer to the *Guardian Procedure Errors and Messages Manual*.

Using Guardian File Names

Guardian file names include disk file names, device names (such as a printer or terminal name), and process file names. You might need to convert the parts of your program that declare and use these names.

Disk File Names. Your existing program might declare a Guardian disk-file-name data item. The largest C-series fully qualified disk file name is 34 characters:

```
* Fully qualified C-series disk file name.

 01  C-DISK-FILE-NAME             PIC X(34).
```

When accessing disk files on other D-series systems in a network, a converted COBOL85 program can use a remote D-series file name with an eight-character volume name (seven characters after the dollar sign) in:

- ☐ A SELECT clause in the FILE-CONTROL paragraph
- ☐ A FILE clause in the SPECIAL-NAMES paragraph

However, a converted COBOL85 program cannot use a remote D-series file name with an eight-character volume name:

- ☐ In a USING or GIVING phrase in a SORT or MERGE statement
- ☐ For a file using fast I/O
- ☐ For spooler job file names

Because of this extra character in the volume name, the largest D-series fully qualified disk file name is 35 characters. You might need to declare your network file-name data items large enough to include this extra character. For example:

```
* Fully qualified D-series disk file name.

 01  DISK-FILE-NAME             PIC X(35)
     VALUE "\NEWYORK.$USERDSK.JUNE1990.REPORTS2".
```

Device Names. Your existing program might declare a Guardian device name. The largest C-series device name without a system name or a qualifier is eight characters (a dollar sign and one to seven characters). The largest C-series network device name without a qualifier is 15 characters (a backslash and a system name followed by a period, a dollar sign, and one to six characters).

When accessing devices on other D-series systems in a network, a converted program can use an eight-character remote device name (one to seven characters after the dollar sign). Therefore, you might need to declare your network device names large enough to include this extra character. For example:

```
* Network D-series device name without a qualifier.

 01  DEVICE-NAME               PIC X(17)
     VALUE "\HAMBURG.$PRTR004".

* Network D-series device name with a qualifier.

 01  DEVICE-NAME-QUALIFIER     PIC X(35)
     VALUE "\HAMBURG.$SPOOLER.#LASRPRT".
```

Process File Names. Your existing program might declare a variable for a C-series Guardian process file name. The D-series operating system uses D-series process file names instead of C-series process file names. Use C-series process file names for compatibility with unconverted C-series procedures.

A D-series process file name is a variable-length string data item with its length specified as a separate data item. A sample declaration follows:

```
* D-series process file name declaration.

01 PROCESS-STUFF.
   03 PROC-NAME-LEN USAGE NATIVE-2.
   03 PROCESS-FILENAME.
      05 P-F OCCURRS 1 TO 47 TIMES DEPENDING ON PROC-NAME-LEN
```

The ZSYSCOB file also contains declarations that you can use for declaring D-series process file name data items.

Declaring Process Identifiers

Your existing program might declare a process-ID data item to identify a process (for example, to represent an opener in an opener table):

```
* Process-ID declaration.

 01 PROCESS-ID.
    05 PROCESS-NAME    PIC X(6).
    05 CPU-PIN         PIC S9(5) COMPUTATIONAL.
```

Convert the process-ID variable declaration to a process-handle variable for process-control operations or to a process-descriptor variable for returning information from a Guardian procedure. Use a process-ID variable for compatibility with unconverted C-series procedures.

A process handle has a 10-word (20-byte) fixed-length structure. A process descriptor is a specific form of the process file name that always includes a node name and sequence number. Examples of declarations are:

```
* D-series process-handle declaration.

 01 PROCESS-HANDLE               PIC X(20).

* D-series process-descriptor declaration.

 01 PROCESS-DESCRIPTOR           PIC X(33).
```

The ZSYSCOB file also contains declarations that you can use for declaring process-handle and process-descriptor data items.

Avoiding Subvolume Defaulting

Your existing program might use subvolume defaulting to represent a Guardian disk file name in the form *volume.file-id*. For example, this ASSIGN clause uses subvolume defaulting to specify the file named MASTER on the $DATA disk volume:

```
* Subvolume defaulting is allowed in a program running
* under the C-series operating system.

 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT MASTER-FILE
     ASSIGN TO "$DATA.MASTER".
```

If you are using the D-series programmatic interface, avoid subvolume defaulting. If a file name requires the volume name, it must also include the subvolume name.

For example:

```
* Subvolume defaulting is NOT allowed in a program running
* under the D-series operating system.

 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT MASTER-FILE
     ASSIGN TO "$DATA.FY90.MASTER".
```

Converting Guardian Procedure Calls

Guardian procedures that you might need to convert are procedures that accept or return:

- ☐ A PIN parameter for a high-PIN process
- ☐ A process-ID parameter

D-series procedures use a process handle (which includes the CPU and PIN values) rather than a process ID to identify a process.

If you convert your program to run at a high PIN, then you must replace MYPID procedure calls with calls to the PROCESS_GETINFO_ and PROCESSHANDLE_DECOMPOSE_ procedures.

"Converting a COBOL85 Program to Run at a High PIN" later in this section, describes most of the Guardian procedures that need converting.

Convert each C-series procedure to the appropriate D-series procedure. The *COBOL85 Programmer's Guide*, the *COBOL85 Reference Manual*, and the *Common Run-Time Environment Programmer's Guide* provide information about converting TAL procedure calls to COBOL85.

Converting for New Reserved Words

FUNCTION is a new reserved word introduced in the D-series COBOL85 compiler. If you have used the word FUNCTION as a variable name you must replace the name with a nonreserved word.

Running the COBOL85 Compiler

When you start the COBOL85 compiler using the TACL RUN command, TACL calls the PROCESS_CREATE_ procedure to create the COBOL85 compiler process and the resulting BINSERV and SYMSERV processes at low PINs.

To run the COBOL85 compiler process (and the BINSERV and SYMSERV processes) at a high PIN, you must use the Binder program to set the HIGHPIN object-file attribute to ON in the COBOL85 compiler object file (provided you have the proper authority to change this file). TACL then runs the COBOL85 compiler (and the BINSERV and SYMSERV processes) at a high PIN if one is available.

Note

When running the COBOL85 compiler at a high PIN, there are restrictions on accessing files that reside on C-series systems for input or output . The compiler returns diagnostic messages if you cannot access a given C-series file. See Appendix C, "System Compatibility," for details.

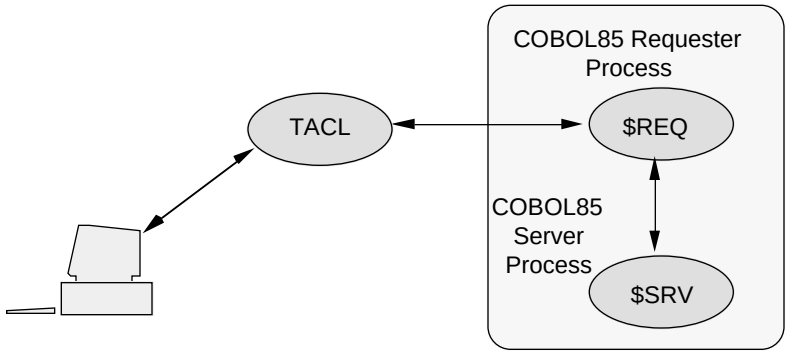
Using the Binder With Converted Object Files

You cannot bind modules that have been compiled with the D-series compiler with modules that have been compiled with a C-series compiler. If you compile any module with the D-series compiler, you must also recompile any other modules that you bind it with.

Converting a COBOL85 Program to Run at a High PIN

This subsection describes how to convert your COBOL85 program to run at a high PIN under the D-series operating system. Figure 4-2 shows a typical application. The box shows which processes this part of the conversion applies to. Converting a COBOL85 program to run at a high PIN applies to both of these processes.

Figure 4-2. Converting a COBOL85 Program to Run at a High PIN



To convert your program, you must:

- ☐ Select the Common Run-Time Environment (CRE).
- ☐ Set the HIGHPIN object-file attribute (which tells the system that your program can run at a high PIN).
- ☐ Make sure that each library file your program uses also has its HIGHPIN object-file attribute set (and is capable of running at a high PIN).
- ☐ Declare PIN data items large enough to hold high-PIN values.
- ☐ Convert calls to the COBOL85^COMPLETION and COBOLSPOOLOPEN utility routines into calls to the COBOL_COMPLETION_ and COBOL_SPECIAL_OPEN_ routines.
- ☐ Remove all ARMTRAP procedure calls.
- ☐ Convert calls to the MYPID Guardian procedure to calls into the PROCESS_GETINFO_ and PROCESSHANDLE_DECOMPOSE_ procedures.

These topics are described in the following subsections.

Selecting the Common Run-Time Environment (CRE)

To convert your COBOL85 program to run at a high PIN, you must select the Common Run-Time Environment (CRE) using the ENV compiler directive.

To select the CRE, specify the COMMON or LIBRARY option for the ENV compiler directive. Place this directive before any source code lines in your program:

```
?ENV COMMON
```

The COMMON or LIBRARY option selects the CRE for the object file. The object file is then compatible with other CRE object files (that is, other COBOL85, TAL, C, Pascal, and FORTRAN object files that are compiled to run with the CRE). TAL object files that have been recompiled with the D-series TAL compiler are also compatible even if they are not compiled to run with the CRE.

The LIBRARY option also allows the object file to be placed in a user library.

The OLD option (the default option) causes the object file to be compatible with C-series object files and other D-series object files that are compiled with the OLD option specified.

The CRE allows a COBOL85 program to call and be called by other programs written in COBOL85, C, Pascal, and FORTRAN, even if the main program is not written in COBOL85. Thus, the CRE has these advantages:

- A program can use the language that is best suited for a specific task. For example, it can perform I/O operations in COBOL85 and then call a C or Pascal routine to perform mathematical computations.
- Routines of a program can share the standard input file, standard output file, and execution log file even though the routines are written in different languages. COBOL85 and FORTRAN routines can also share the same $RECEIVE file.
- Error reporting and exception handling are provided in the CRE routines.

For more information about the CRE, including the requirements for binding COBOL85, TAL, C, Pascal, and FORTRAN object files using Binder, refer to the *Common Run-Time Environment (CRE) Programmer's Guide.*

Setting the HIGHPIN Object-File Attribute

The HIGHPIN object-file attribute directs the system to run a program at a high PIN if one is available. If a high PIN is not available, the program runs at a low PIN if one is available. You set the HIGHPIN object-file attribute either during compilation using a compiler directive or after compilation using the Binder program.

To set the attribute when you compile your program, specify the HIGHPIN directive in your source code or as a compiler option in the TACL RUN command for the COBOL85 compiler. The BINSERV program then sets the HIGHPIN attribute in the object file. An example of this directive (with the ENV directive) in a source file is:

```
?ENV COMMON
?HIGHPIN
```

An example of this directive as a compiler option is:

```
10> COBOL85 / ... / cobobj; ENV COMMON; HIGHPIN
```

You need to specify the HIGHPIN directive only once during a compilation. However, you can specify it any number of times and the compiler will not generate an error.

Note If you compile with the HIGHPIN directive but do not specify COMMON or LIBRARY for the ENV directive, the COBOL85 compiler generates a warning message and ignores the HIGHPIN directive.

If you do not set the HIGHPIN attribute when you compile your program, you can set it after compilation using the Binder program.

If you are binding more than one object file into a single target object file, the Binder program sets the HIGHPIN object-file attribute in the target file only if all constituent files have the HIGHPIN object-file attribute set. If necessary, use the Binder CHANGE command to set the attribute in the target object file:

```
@CHANGE HIGHPIN ON IN cobobj
```

Using a Library File

If your existing program uses a library file, the library file must also have its HIGHPIN object-file attribute set. To determine the current setting of the HIGHPIN attribute for a library file, use the Binder SHOW command:

```
@SHOW SET HIGHPIN FROM libfile
```

If necessary, set this attribute as described in the previous step (provided the library file has been converted to support a high-PIN process).

Declaring CPU and PIN Data Items

As already stated under "Declaring and Using Guardian Data Items," earlier in this section, your existing program might declare either a two-digit data item for the CPU value and a three-digit data item for the PIN value, or a three-digit data item for just a PIN value:

```
WORKING-STORAGE SECTION.
01 CPU-PIN-DEFINITIONS.
   05 CPU             PIC S9(2) COMPUTATIONAL.
   05 PIN             PIC S9(3) COMPUTATIONAL.
```

Use a USAGE IS NATIVE-2 clause in the declaration of all PIN values, including PINs for backup processes, to allow up to 32,767 PINs. Declare a CPU number as a separate two-digit unsigned data item:

```
WORKING-STORAGE SECTION.
01 CPU-PIN-DEFINITIONS.
   05 CPU             PIC S9(2) COMPUTATIONAL.
   05 PIN             NATIVE-2.
```

Note that you can declare each PIN using a PIC 9(4) COMPUTATIOSAL clause if you will never need a PIN higher than 9,999.

Calling COBOL85 Utility Routines

If your existing program calls utility routines in the COBOLLIB or CBL85UTL library files, you might need to convert these calls to the appropriate D-series routines. The routines that you must convert are shown in Table 4-1. For a description of all COBOL85 utility routines, refer to the *COBOL85 Reference Manual*.

Table 4-1. COBOL85 Utility Routines

C-Series Routine (ENV is OLD or omitted)	D-Series Routine (ENV is COMMON or LIBRARY)	Reason for Conversion
COBOL85^COMPLETION	COBOL_COMPLETION_	The COBOL85^COMPLETION routine *text-length* parameter has been removed.
COBOLSPOOLOPEN	COBOL_SPECIAL_OPEN_	The COBOLSPOOLOPEN routine has been deleted.

Removing ARMTRAP Procedure Calls

If your existing program calls the ARMTRAP Guardian procedure using the ENTER statement, remove all calls to this procedure. If you specify the CRE, CRE routines perform all trap handling. (You are not required to remove calls to the COBOL85^ARMTRAP routine; however, it is no longer needed because the CRE selects a trap automatically.)

Converting MYPID Procedure Calls

Your existing program might call the MYPID Guardian procedure using the ENTER statement to obtain its CPU and PIN values:

```
WORKING-STORAGE SECTION.
01  CPU-PIN             PIC S9(5) COMPUTATIONAL.
...
PROCEDURE DIVISION.
...

ENTER TAL "MYPID" GIVING CPU-PIN
```

If a high-PIN process calls MYPID, a trap condition occurs. Convert MYPID procedure calls into PROCESSHANDLE_DECOMPOSE_ procedure calls.

The PROCESSHANDLE_DECOMPOSE_ procedure requires a process handle as an input parameter. If you do not know the process handle of your process, first call the PROCESSHANDLE_GETMINE__ procedure. Then pass the results to PROCESSHANDLE_DECOMPOSE_ , which returns the CPU and PIN values as separate integer values. For example:

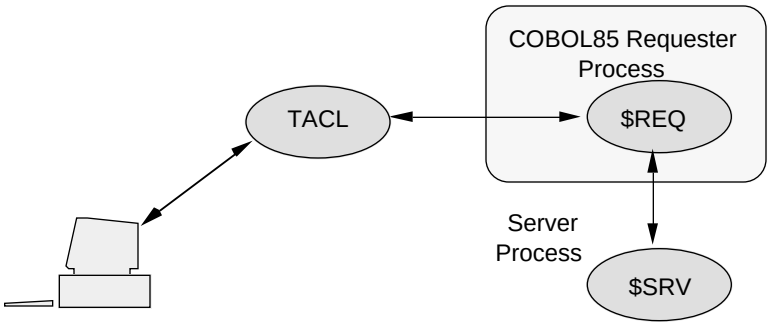
```
WORKING-STORAGE SECTION.
01  PROCESS-HANDLE       PIC X(20).
01  CPU-PIN.
    05  CPU              PIC S9(2) COMPUTATIONAL.
    05  PIN              NATIVE-2.
01  ERROR-NUMBER         PIC S9(5) COMPUTATIONAL.
01  NULL-PH              PIC X(20) VALUE ALL HIGH-VALUES.
...

PROCEDURE DIVISION.
...

ENTER TAL "PROCESSHANDLE_GETMINE_"
                      USING PROCESS-HANDLE
                      GIVING ERROR-NUMBER

ENTER TAL "PROCESSHANDLE_DECOMPOSE_"
                      USING PROCESS-HANDLE
                      CPU
                      PIN
                      GIVING ERROR-NUMBER
```

Creating and Managing a High-PIN Process

This subsection describes how to convert a program that creates and manages a new process. The converted program uses the D-series enhanced interface. The process you create must already be converted to run at a high PIN as described under "Converting a COBOL85 Program to Run at a High PIN" earlier in this section.

Figure 4-3 shows the processes involved in converting a typical application. The steps described in this subsection apply to the requester process $REQ, which creates the high-PIN server process $SRV.

Figure 4-3. Converting a COBOL85 Program to Create and Manage a High-PIN Process



This subsection covers the following topics:

- Programmatically creating a new process at a high PIN
- Managing a high-PIN process, including activating, suspending, stopping, and abending the process, as well as invoking Inspect or Debug for the process

For information on how to interactively create a high-PIN process using TACL, refer to Section 7, "Converting TACL Programs."

Creating a High-PIN Process

If you have already converted your program to run at a high PIN as described earlier in this section under "Converting a COBOL85 Program to Run at a High PIN," then your program will automatically create any new processes at a high PIN, assuming:

- ☐ The process you are creating is designed to run at a high PIN (that is, it has its HIGHPIN object-file attribute set). See "Setting the HIGHPIN Object-File Attribute" earlier in this section.
- ☐ The program you are converting is not started by another low-PIN process that has its inherited force-low characteristic set. See Appendix C, "System Compatibility," for a discussion of the inherited force-low characteristic.

Starting a new process at a high PIN is automatic because compiling with the CRE forces the object file to make Guardian procedure calls through the D-series enhanced interface rather than the C-series-compatible interface.

See "Selecting the Common Run-Time Environment (CRE)" earlier in this section for details on how to compile with the CRE.

Managing a High-PIN Process

Managing a process can involve suspending and activating the process, and stopping or abending the process. To perform these operations on a high-PIN process, you simply need to recompile your COBOL85 program to select the CRE. Doing so causes the object code to use the D-series interface to the operating system, which is necessary to manage a process at a high PIN.

See "Selecting the Common Run-Time Environment (CRE)" earlier in this section for details on how to compile with the CRE.

Opening and Communicating With a High-PIN Server

This subsection describes how to convert a COBOL85 requester to communicate with and monitor a high-PIN server. For the requester, you have the following options:

- ☐ The requester remains unconverted, but the server has the RUNNAMED object-file attribute set.
- ☐ The requester is converted to run at a high PIN and its interface with the server process is also converted for communication with a high-PIN process.

If your requester does not need converting, then you can let it continue to run at a low PIN by setting the RUNNAMED object-file attribute in the server. This attribute makes sure that the server is named and therefore allows the requester to open it even though the requester runs at a low PIN and the server at a high PIN. "Setting the RUNNAMED Object-File Attribute" later in this subsection explains how to do this.

If your requester does need to be converted, you must first make it run at a high PIN as described in "Converting a COBOL85 Program to Run at a High PIN" earlier in this section. Then, if necessary, convert the following parts of the program:

- ☐ Communication with the server
- ☐ Server monitoring functions
- ☐ System message reception

See "Converting a Requester," later in this section, for details on how to convert these parts of your program.

For information about converting a server to monitor a high-PIN requester process, including maintaining an opener table, refer to "Being Opened by and Communicating With a High-PIN Requester" later in this section.

Setting the RUNNAMED Object-File Attribute

The RUNNAMED object-file attribute causes a process to run as a named process even if you do not provide a name for it. Because you can then open the process by name, you can run the process at a high PIN under the D-series operating system and have that process opened by an unconverted process using the Guardian OPEN procedure.

For more information about how an unconverted process running on a D-series system can communicate with a named high-PIN process, refer to Appendix C, "System Compatibility."

You set the RUNNAMED object-file attribute either during compilation using a compiler directive or after compilation using the Binder program.

To set the attribute when you compile your program, specify the RUNNAMED compiler directive in your source code or as a compiler option in the TACL RUN command for the COBOL85 compiler. The BINSERV program then sets the RUNNAMED attribute in the object file. An example of this directive (with the ENV and HIGHPIN directives) in a source file is:

```
?ENV COMMON; HIGHPIN; RUNNAMED
```

An example of this directive as a compiler option is:

```
10> COBOL85 / ... / cobobj; ENV COMMON; HIGHPIN; RUNNAMED
```

You need only specify the RUNNAMED directive once during a compilation. However, you can specify it any number of times and the compiler will not generate an error.

If you do not set the RUNNAMED attribute when you compile your program, you can set it after compilation using Binder. For a single object file, use the Binder CHANGE command:

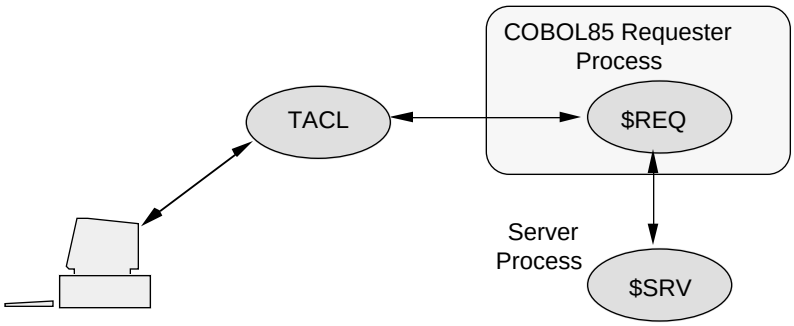
```
@CHANGE RUNNAMED ON IN cobobj
```

If you are binding more than one object file into a single target object file, use the Binder SET command to set the object-file attribute. If any of the constituent object files used to build the target file has the RUNNAMED object-file attribute set, Binder sets the attribute in the target object file.

Converting a Requester

This subsection describes how to convert your COBOL85 requester program. Figure 4-4 shows the processes involved in converting a typical application. The steps in this subsection apply to the requester process $REQ.

Figure 4-4. Converting a COBOL85 Requester to Communicate With a High-PIN Server



This subsection describes the following topics:

- ☐ Opening and communicating with a high-PIN server
- ☐ Monitoring a High-PIN server
- ☐ Opening and reading $RECEIVE
- ☐ Converting the RECEIVE-CONTROL Paragraph

Opening and Communicating With a High-PIN Server

To use the D-series enhanced interface for opening and communicating with a high-PIN server, you simply need to recompile your COBOL85 program using the CRE.

See "Selecting the Common Run-Time Environment (CRE)" earlier in this section for details on how to compile with the CRE.

Monitoring a High-PIN Server

If your requester program monitors a server, then it must read and handle process-deletion and status-change messages. To do so for a high-PIN server, your program must:

- ☐ Declare a read buffer large enough for D-series enhanced system messages.
- ☐ Check for and process the new D-series process-deletion and status change messages.

The following paragraphs describe these operations.

Once again, to receive the D-series system messages, your program must be recompiled with the CRE.

Opening and Reading $RECEIVE

You might need to convert the part of your program that opens and reads $RECEIVE as follows:

1. Your OPEN statement for $RECEIVE should not require any changes.
2. When you read a D-series message from $RECEIVE using the READ statement, use a message record area that is at least 256 characters in length.

   The ZSYSCOB file contains a data structure for each system message. Because each system-message data structure is in a separate section of ZSYSCOB, you can copy only the data structures for the messages you actually read.
3. If necessary, modify the parts of your program that process each system message. Refer to Table 4-2 for the D-series system messages that supersede C-series system messages.
4. Your CLOSE statement for $RECEIVE should not require any changes.

Converting the RECEIVE-CONTROL Paragraph

The RECEIVE-CONTROL paragraph allows a program to read system messages by defining the receive-control and reply tables. The parts of the RECEIVE-CONTROL paragraph that you might need to convert are:

- ☐ The REPORT clause, which specifies the system messages a program reads from $RECEIVE
- ☐ The MESSAGE SOURCE clause, which identifies the process that sent a system message

The REPORT Clause. The REPORT clause in the RECEIVE-CONTROL paragraph specifies the specific system messages that your program reads from $RECEIVE.

You might need to convert this clause to specify the D-series messages you want to read from $RECEIVE.

Table 4-2 shows the REPORT clause message-type keywords and the corresponding C-series and D-series system messages that each keyword allows a program to read from $RECEIVE.

Table 4-2. Message-Type Keywords (Page 1 of 2)

COBOL85 Keyword	C-Series System Message (ENV is OLD or omitted)		D-Series System Message (ENV is COMMON or LIBRARY)	
ABEND	-6	Process abnormal deletion: Abend		Not used; see PROCESS-DELETION
BREAK	-20	Break on device	-105	Break on device
CLOSE	-31	Process close	-104	Process close
CONTROL	-32	Process CONTROL	-32	Process CONTROL
CONTROLBUF	-35	Process CONTROLBUF	-35	Process CONTROLBUF
CPU-DOWN	-2 -2	CPU down: process MONITORCPUS CPU down: named process deletion	-2 -101	CPU down: process MONITORCPUS Process deletion for CPU down
CPU-UP	-3	CPU up	-3	CPU up
DEVICE-INFO	-40	Device type inquiry	-106	Device type inquiry
DEVICEINFO2-COMPLETION	-41	Nowait device type inquiry	-41	Nowait device type inquiry
FILE-GETINFOBYNAME-COMPLETION		None	-108	Nowait FILE_GETINFOBYNAME_ completion
FILENAME-FILENEXT-COMPLETION		None	-109	Nowait FILENAME_FINDNEXT_ completion
JOB-PROCESS-CREATION	-9	Job process creation	-112	Job process creation
LOGICAL-CLOSE		Logical open		Logical open
LOGICAL-OPEN		Logical open		Logical open
MEMORY-LOCK-COMPLETION	-23	Memory lock completion	-23	Memory lock completion
MEMORY-LOCK-FAILURE	-24	Memory lock failure	-24	Memory lock failure
MESSAGE-CANCELLED	-38	Queued message cancellation	-38	Queued message cancellation
MESSAGE-MISSED	-13	System message buffer overrun	-13	System message buffer overrun
NETWORK	-8	Change in status of network node		Not used; see NODE-DOWN, NODE-UP, REMOTE-CPU-DOWN, and REMOTE-CPU-UP
NEWPROCESSNOWAIT-COMPLETION or NEWPROCESS-COMPLETION	-12	NEWPROCESSNOWAIT completion	-12	NEWPROCESSNOWAIT completion

Table 4-2. Message-Type Keywords (Page 2 of 2)

COBOL85 Keyword	C-Series System Message (ENV is OLD or omitted)		D-Series System Message (ENV is COMMON or LIBRARY)	
NODE-DOWN	-8	Change in status of network node	-110	Loss of communication with node
NODE-UP	-8	Change in status of network node	-111	Establishment of communication with node
OPEN	-30	Process open	-103	Process open
POWER-ON	-11	Power on	-11	Power on
PROCESS-CREATE-CREATION		None	-102	Nowait PROCESS_CREATE_ completion
PROCESS-DELETION	-2 -5 -6	CPU down: named process deletion Stop Abend	-101	Process deletion: STOP, ABEND, or CPU down
PROCESS-TIME-SIGNAL	-26	Process time timeout	-26	Process time timeout
REMOTE-CPU-DOWN	-8	Change in status of network node	-100	Remote CPU down
REMOTE-CPU-UP	-8	Change in status of network node	-113	Remote CPU up
RESETSYNC	-34	Process RESETSYNC	-34	Process RESETSYNC
SETMODE	-33	Process SETMODE	-33	Process SETMODE
SETPARAM	-37	Process SETPARAM	-37	Process SETPARAM
SETTIME	-10	SETTIME	-10	SETTIME
STATUS-3270	-21	3270 device status received	-21	3270 device status received
STOP	-5	Process normal deletion: Stop		Not used; see PROCESS-DELETION
SUBORDINATE-NAME		None	-107	Subordinate name inquiry
SYSTEM		Specifies system messages –5, –6, –12, –22, –23, –24, –30, –31, –32, –33, –34, and –35		Specifies all system messages except LOGICAL-OPEN and LOGICAL-CLOSE
TIME-SIGNAL	-22	Elapsed time timeout	-22	Elapsed time timeout

The following paragraphs identify the system messages that are most likely affected by the conversion of a requester. These messages include the process-deletion system messages and the status-change system messages.

Reading Process-Deletion System Messages. Your requester might monitor a server process by using the following keywords to read process-deletion system messages from $RECEIVE:

COBOL85 Keyword	C-Series System Message (ENV is OLD or Omitted)	
CPU-DOWN	-2	CPU down: named process deletion
STOP	-5	Process normal deletion: Stop
ABEND	-6	Process abnormal deletion: Abend
PROCESS-DELETION	-2	CPU down: named process deletion
	-5	Process normal deletion: Stop
	-6	Process abnormal deletion: Abend

Convert your requester to use the PROCESS-DELETION keyword to read and process message -101 (Process deletion), which supersedes all the above messages:

COBOL85 Keyword	D-Series System Message (ENV is COMMON or LIBRARY)	
PROCESS-DELETION	-101	Process deletion

Reading Status-Change System Messages. Your requester might monitor a server process using the following keywords to read status-change system messages from $RECEIVE:

COBOL85 Keyword	C-Series System Message (ENV is OLD or Omitted)	
CPU-DOWN	-2	Process MONITORCPUS
NETWORK	-8	Change in status of network node
NODE-DOWN	-8	Change in status of network node
NODE-UP	-8	Change in status of network node
REMOTE-CPU-DOWN	-8	Change in status of network node
REMOTE-CPU-UP	-8	Change in status of network node

Convert your program as follows:

- ☐ Continue to use the CPU-DOWN keyword to read system message -2.
- ☐ Replace the use of the NETWORK keyword with one of NODE-DOWN, NODE-UP, REMOTE-CPU-DOWN, or REMOTE-CPU-UP, which now correspond to new system messages as shown in the table below.
- ☐ Continue with your current use of keywords NODE-DOWN, NODE-UP, REMOTE-CPU-DOWN, and REMOTE-CPU-UP, but note that these keywords now read the new D-series system messages as listed in the following table:

COBOL85 Keyword	D-Series System Message (ENV is COMMON or LIBRARY)	
CPU-DOWN	-2	Process MONITORCPUS
NODE-DOWN	-110	Loss of communication with node
NODE-UP	-111	Establishment of communication with node
REMOTE-CPU-DOWN	-100	Remote CPU down
REMOTE-CPU-UP	-113	Remote CPU up

The MESSAGE SOURCE Clause. If your existing program receives system messages, the MESSAGE SOURCE clause in the RECEIVE-CONTROL paragraph identifies the process that sent the message. The C-series MESSAGE SOURCE clause uses a process ID to identify the sender process:

```
* C-series receive-control table.

 RECEIVE-CONTROL.
   TABLE OCCURS 1 TIMES
   SYNCDEPTH LIMIT IS 1
   MESSAGE SOURCE IS MESSAGE-SENDER.

...

 01 MESSAGE-SENDER.
   05  SYSTEM-FLAG       PIC S9  COMPUTATIONAL.
   05  ENTRY-NUMBER      PIC 999 COMPUTATIONAL.
   05  FILLER            PIC X(4).
   05  PROCESS-ID.
     10 PROCESS-NAME     PIC X(6).
     10 CPU-PIN          PIC S9(2) COMPUTATIONAL.
   05  FILLER            PIC X(16).
```

Convert your existing program to identify the sender process using a process handle rather than a process ID:

```
* D-series receive-control table.

 RECEIVE-CONTROL.
   TABLE OCCURS 1 TIMES
   SYNCDEPTH LIMIT IS 1
   MESSAGE SOURCE IS MESSAGE-SENDER.

...

 01 MESSAGE-SENDER.
   05  SYSTEM-FLAG       PIC S9  COMPUTATIONAL.
   05  ENTRY-NUMBER      PIC 999 COMPUTATIONAL.
   05  FILLER            PIC X(4).
   05  MSG-SOURCE        PIC X(20).
   05  FILLER            PIC X(4).
```

After conversion, the overall length of the MESSAGE SOURCE data item is the same (32 characters), but the process-handle declaration replaces the process-ID declaration. If you need values from the process-handle data item (for example, the CPU and PIN values), use the PROCESSHANDLE_DECOMPOSE_ procedure.

Being Opened by and Communicating With a High-PIN Requester

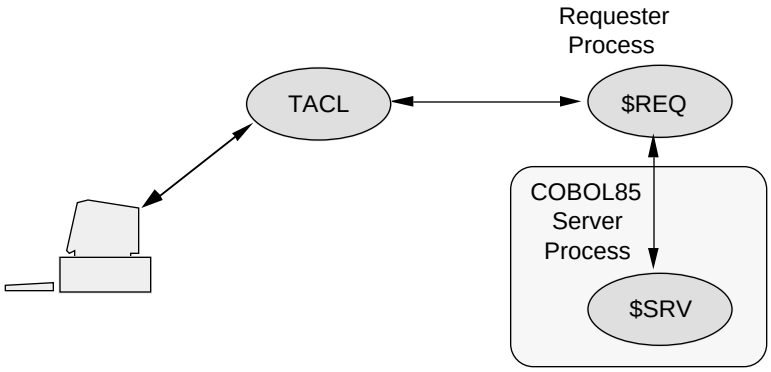
This subsection describes how to convert a COBOL85 server to communicate with a high-PIN requester. Whether you need to convert the server process depends in part on whether the server tracks its openers. If the server does keep track of its openers, you should enable the server to run at a high PIN as described in "Converting a COBOL85 Program to Run at a High PIN," earlier in this section, and then convert the server as described under "Converting a Server," later in this subsection.

If the server does not track its openers, or if you choose not to perform the conversion, then you can keep the server process at a low PIN and not convert it, except for setting the HIGHREQUESTERS object-file attribute as described under "Setting the HIGHREQUESTERS Attribute to Allow High-PIN Openers," later in this subsection. Setting this attribute enables a high-PIN requester to open a low-PIN server.

Converting a Server

If your server program tracks its openers and uses the PIN to identify each opener, then you must convert the server. Figure 4-5 shows the processes involved in converting a typical application. The steps described in this subsection apply to the server process $SRV.

Figure 4-5. Converting a COBOL85 Server to Communicate With a High-PIN Requester



You must convert the following parts of your program:

- ☐ The RECEIVE-CONTROL paragraph, so that
  - ☐ Your program accepts D-series enhanced interface messages instead of C-series-compatible messages
  - ☐ The MESSAGE SOURCE clause identifies processes by process handle rather than by process ID
- ☐ The opener table, so that it identifies openers by process handle instead of process ID
- ☐ $RECEIVE handling, to have a read buffer large enough for D-series enhanced interface system messages

Converting the RECEIVE-CONTROL Paragraph

The RECEIVE-CONTROL paragraph allows a program to read system messages by defining the receive-control and reply tables. The parts of the RECEIVE-CONTROL paragraph that you might need to convert are:

- The REPORT clause, which specifies the system messages a program reads from $RECEIVE
- The MESSAGE SOURCE clause, which identifies the process that sent a system message

The REPORT Clause. The REPORT clause in the RECEIVE-CONTROL paragraph specifies the specific system messages that your program reads from $RECEIVE.

You might need to convert this clause to specify the D-series messages you want to read from $RECEIVE.

Table 4-2 shows the REPORT clause message-type keywords and the corresponding C-series and D-series system messages that each keyword allows a program to read from $RECEIVE.

The following paragraphs identify the system messages that are most likely affected by converting a server. These messages include the Open and Close system messages and the status-change system messages.

Reading Open and Close System Messages. If your server maintains an opener table, then it will read all Open and Close system messages. Your existing program uses the following keywords to read these messages:

COBOL85 Keyword	C-Series System Message (ENV is OLD or Omitted)	
OPEN	-30	Process open
CLOSE	-31	Process close

For these keywords, no conversion is necessary because the same keywords are used to read the corresponding system messages generated by the D-series enhanced interface when you recompile your program with CRE. The correspondence between the keywords and the system messages is shown below:

COBOL85 Keyword	D-Series System Message (ENV is COMMON or LIBRARY)	
OPEN	-103	Process open
CLOSE	-104	Process close

Reading Status-Change System Messages. Your server might also monitor the status of its requesters by using the following keywords to read status-change system messages from $RECEIVE:

COBOL85 Keyword	C-Series System Message (ENV is OLD or Omitted)	
CPU-DOWN	-2	Process MONITORCPUS
NETWORK	-8	Change in status of network node
NODE-DOWN	-8	Change in status of network node
NODE-UP	-8	Change in status of network node
REMOTE-CPU-DOWN	-8	Change in status of network node
REMOTE-CPU-UP	-8	Change in status of network node

Convert your program as follows:

- ☐ Continue to use the CPU-DOWN keyword to read system message -2.
- ☐ Replace the use of the NETWORK keyword with one of NODE-DOWN, NODE-UP, REMOTE-CPU-DOWN, or REMOTE-CPU-UP, which now correspond to new system messages as shown in the table below.
- ☐ Continue with your current use of keywords NODE-DOWN, NODE-UP, REMOTE-CPU-DOWN, and REMOTE-CPU-UP, but note that these keywords now read the new D-series system messages as listed in the following table.

COBOL85 Keyword	D-Series System Message (ENV is COMMON or LIBRARY)	
CPU-DOWN	-2	Process MONITORCPUS
NODE-DOWN	-110	Loss of communication with node
NODE-UP	-111	Establishment of communication with node
REMOTE-CPU-DOWN	-100	Remote CPU down
REMOTE-CPU-UP	-113	Remote CPU up

The MESSAGE SOURCE Clause. If your existing program receives system messages, the MESSAGE SOURCE clause in the RECEIVE-CONTROL paragraph identifies the process that sent the message. The C-series MESSAGE SOURCE clause uses a process ID to identify the sender process:

```
* C-series receive-control table.

 RECEIVE-CONTROL.
   TABLE OCCURS 1 TIMES
   SYNCDEPTH LIMIT IS 1
   MESSAGE SOURCE IS MESSAGE-SENDER.

...

 01 MESSAGE-SENDER.
   05  SYSTEM-FLAG       PIC S9  COMPUTATIONAL.
   05  ENTRY-NUMBER      PIC 999 COMPUTATIONAL.
   05  FILLER            PIC X(4).
   05  PROCESS-ID.
     10 PROCESS-NAME     PIC X(6).
     10 CPU-PIN          PIC S9(2) COMPUTATIONAL.
   05  FILLER            PIC X(16).
```

Convert your existing program to identify the sender process using a process handle rather than a process ID:

```
* D-series receive-control table.

 RECEIVE-CONTROL.
   TABLE OCCURS 1 TIMES
   SYNCDEPTH LIMIT IS 1
   MESSAGE SOURCE IS MESSAGE-SENDER.

...

 01 MESSAGE-SENDER.
   05  SYSTEM-FLAG       PIC S9  COMPUTATIONAL.
   05  ENTRY-NUMBER      PIC 999 COMPUTATIONAL.
   05  FILLER            PIC X(4).
   05  MSG-SOURCE        PIC X(20).
   05  FILLER            PIC X(4).
```

After conversion, the overall length of the MESSAGE SOURCE data item is the same (32 characters), but the process-handle declaration replaces the process-ID declaration. If you need values from the process-handle data item (for example, the CPU and PIN values), use the PROCESSHANDLE_DECOMPOSE_ procedure.

Defining an Opener Table

If your program is a server that tracks its openers, you might define an opener table using a process ID to identify an opener:

```
* C-series opener table.

 01 OPENER-TABLE.
    03 OPENERS OCCURS 15 TIMES.
       05 CURRENT-COUNT     PIC 999 COMPUTATIONAL.
       05 PRIMARY-OPENER.
          07 PROCESS-NAME      PIC X(6).
          07 CPU-PIN           PIC S9(5).
       05 BACKUP-OPENER.
          07 PROCESS-NAME      PIC X(6).
          07 CPU-PIN           PIC S9(5).
```

Convert your opener table to identify an opener using a process handle rather than a process ID. For example:

```
* D-series opener table.

 01 OPENER-TABLE.
    03 OPENERS OCCURS 15 TIMES.
       05 CURRENT-COUNT     PIC 999 COMPUTATIONAL.
       05 PRIMARY-OPENER    PIC X(20).
       05 BACKUP-OPENER     PIC X(20).
```

Opening and Reading $RECEIVE

You might need to convert the part of your program that opens and reads $RECEIVE as follows:

1. Your OPEN statement for $RECEIVE should not require any changes.
2. When you read a D-series system message from $RECEIVE using the READ statement, use a message record area that is at least 256 characters in length.

   The ZSYSCOB file contains a data structure for each system message. Because each system-message data structure is in a separate section of ZSYSCOB, you can copy only the data structures for the messages you actually read.
3. If necessary, modify the parts of your program that process each system message. Refer to Table 4-2 for the D-series messages that supersede C-series messages.
4. Your CLOSE statement for $RECEIVE should not require any changes.

Setting the HIGHREQUESTERS Attribute to Allow High-PIN Openers

The HIGHREQUESTERS object-file attribute allows a process to support requests from high-PIN requesters. Use this attribute only for a COBOL85 main program. You can set the HIGHREQUESTERS object-file attribute by including a compiler directive in your source file, or you can set it after you have finished converting your source code either with a compiler option or after compilation using the Binder program.

To set the attribute when you compile your program, specify the HIGHREQUESTERS directive in your source code or as a compiler option in the TACL RUN command for the COBOL85 compiler. The BINSERV program then sets the HIGHREQUESTERS attribute in the object file. An example of this directive in a source file is:

```
?HIGHREQUESTERS
```

An example of this directive as a compiler option is:

```
10> COBOL85 /IN cobsrc, ... / cobobj ; HIGHREQUESTERS
```

You need to specify the HIGHREQUESTERS directive only once during a compilation. If your existing program copies source code from another file, specify the HIGHREQUESTERS directive only in the main program file; do not specify the directive in the other file (or files).

If you do not set the HIGHREQUESTERS attribute when you compile your program, you can set it after compilation using Binder. For a single object file, use the Binder CHANGE command:

```
@CHANGE HIGHREQUESTERS ON IN cobobj
```

If you are binding more than one object file into a single target object file, use the Binder SET command to set the HIGHREQUESTERS object-file attribute. For Binder to set the HIGHREQUESTERS object-file attribute in a target object file, the object file containing the main program must have this object-file attribute set.

For more information about the HIGHREQUESTERS object-file attribute, refer to "Allowing Opens by High-PIN Requesters" in Appendix C, "System Compatibility."

5 Converting C Applications

A C program can run at a low PIN on a D-series operating system without any changes. However, for a C program to use the extended features of the D-series operating system, specific parts of it must be converted.

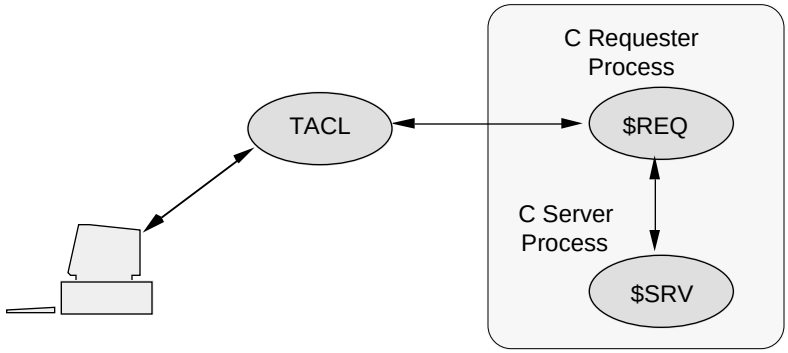
The topics in this section are:

- ☐ Recompiling your C program
- ☐ Program elements affected by D-Series system enhancements
- ☐ Making the C compiler run as a high PIN process
- ☐ Converting a C program to run at a high PIN
- ☐ Converting a C program to create a high-PIN process
- ☐ Converting a requester to open and communicate with a high-PIN server
- ☐ Converting a server to be opened by and communicate with a high-PIN requester

Section 8, “Converting Other Parts of an Application,” contains information about converting other parts of a C application. For additional information about the Tandem implementation of C, refer to the *C Reference Manual.*

Figure 5-1 shows a typical application. The box shows which processes this part of the conversion applies to. Converting basic elements of a C program applies to both of these processes.

Figure 5-1. Converting Basic Elements of a C Program



Recompiling Your C Program

Some elements of a C-series C program must be changed if you recompile the program on a D-series system. To recompile your program, you must:

- ☐ Compile all program modules with a D-series C compiler
- ☐ Use the D-series CEXTDECS file (external declarations for Guardian procedures)
- ☐ Use the D-series ZSYSC file (source declarations for Guardian procedures and system messages)
- ☐ Change memory-model file references in SEARCH compiler pragmas
- ☐ Open temporary files using either the ANSI `tempfile` function or the Guardian FILE_CREATE_ procedure
- ☐ Replace `min` and `max` macros
- ☐ Declare the the NULL macro
- ☐ Change macro definitions that replace parameters inside a literal string
- ☐ Specify a FIELDALIGN SHARED2 pragma for type long used in bit-field declarations
- ☐ Use the new definition for `errno`
- ☐ Change code that relies on the `sizeof` operator returning 2 for a constant operand
- ☐ Change code that relies on `size_t`, the result of the `sizeof` operator, being of type long
- ☐ Change code that uses the `fflush` function to flush partial lines to `stdout` or `stderr`
- ☐ Change code that relies on the return value of the `sscanf` function

Additional elements of a C-series C program must be changed if you want the program to be compliant with ANSI C. You do not have to change these elements at this time. However, Tandem recommends that you change these elements now because future releases of the Tandem C compiler might not support these elements. To make your program ANSI C compliant, you must:

- ☐ Change certain keywords
- ☐ Replace obsolete TAL function declarations
- ☐ Declare a function prototype within the scope of each call to a function

These topics are discussed in this subsection.

D-Series and C-Series Object Modules

You cannot bind modules that have been compiled by a D-series compiler with modules that have been compiled by a C-series compiler. If you compile any module with a D-series compiler, you must also recompile any other modules that you bind with it.

D-Series CEXTDECS Declarations

The $SYSTEM.SYSTEM.CEXTDECS file contains external declarations for Guardian procedures. Some C-series Guardian procedures have been replaced by new D-series procedures. The D-series CEXTDECS file contains declarations for both the C-series version and D-series version of these procedures.

Convert your `#include` compiler directive to specify D-series declarations instead of C-series declarations. Then replace the C-series procedure names in your program with the new D-series procedure names.

For example:

`#include` Specifying Superseded C-Series Procedures	`#include` Specifying D-Series Procedures

`#include <cextdecs (OPEN, \`
`READX, \`
`WRITEX,\`
`CLOSE)>`
	`#include <cextdecs(FILE_OPEN_,\`
	`READX, \`
	`WRITEX, \`
	`FILE_CLOSE_)>`

Appendix A lists the D-series procedures that supersede C-series procedures. For a description of each procedure, refer to the *Guardian Procedure Calls Reference Manual*.

D-Series ZSYSC Declarations

The $SYSTEM.ZSYSDEFS.ZSYSC file contains source declarations for Guardian procedures and system messages.

Note Contact your system manager if you cannot find $SYSTEM.ZSYSDEFS.ZSYSC. The file may have been relocated.

To use these declarations, include them with your source code using the `#include` compiler directive.

The following directive includes the entire ZSYSC file without printing the contents of the file in a compiler listing:

```
#include "$system.zsysdefs.zsysc" nolist
```

The ZSYSC file is divided into sections, which allows you to include only the sections your program actually needs. The following directive includes only the process-creation and system-message constant declarations and prints the contents of each section in a compiler listing:

```
#include "$system.zsysdefs.zsysc (process_constant,          \
                                  system_messages_constant)"
```

To print a listing of the ZSYSC file to check the declarations that are available for your program, use the FUP COPY command:

```
10> FUP COPY $SYSTEM.ZSYSDEFS.ZSYSC, $s.#lineptr
```

Changing Memory-Model File References

If your program uses a SEARCH compiler pragma to specify a memory-model file (to search when resolving external references), you must change the name of the memory-model file as follows:

C-Series `pragma` Format (Superseded)	D-Series `pragma` Format
`#pragma SEARCH smallc`	`#pragma SEARCH csmall`
`#pragma SEARCH largec`	`#pragma SEARCH clarge`
`#pragma SEARCH widec`	`#pragma SEARCH cwide`

Opening Temporary Files

Temporary files must be opened with either the ANSI `tempfile` function or the Guardian FILE_CREATE_ procedure.

C-Series `open` Supplementary Function (Superseded)

```
char temp_file[26] = "$spool                    ";
int status;
status = open (temp_file, O_CREATE | O_BINARY);
```

D-Series FILE_CREATE_ Procedure

```
char temp_file[26] = "$spool";
short error;
short volume_length = 6;
error = FILE_CREATE_ (temp_file, volume_length,
                      filename_length);
```

Replacing `min` and `max` Macros

You must change the following macros in your program:

C-Series Macro (Superseded)	D-Series Macro
`min`	`_min`
`max`	`_max`

Including the Macro NULL Definition

The definition of the object-like macro NULL was removed from the C header files ASSERTH, CTYPEH, ERRNOH, FLOATH, LIMITSH, MATHH, and SETJMPH to make these header files conform to the ANSI standard.

If your program uses the macro NULL, you must include a standard library header file which declares the macro, such as STDIOH.

Changing Macro Definitions

The D-series compiler macro processing has been changed to conform to the ANSI standard. You must change your macros to use a # directive to replace parameters inside a literal string.

The following macro is valid for C-series compilers:

```
#define pr(x, format) printf("The x = %format\n", (x))
```

You must change this macro for D-series compilers as follows:

```
#define pr(x, format) printf("The " #x " = %% " #format \
   "\n", (x))
```

Using Type long in Bit-field Declarations

Programs should only use the type int in bit-field declarations. ANSI C only supports type int for bit field declarations. However, the D-series compiler continues to support bit field declarations of type short for large-memory model and type long for wide-data model. To continue to use type long in bit-field declarations under the wide-data model, you must specify a FIELDALIGN SHARED2 pragma for the module that contains the declarations. The pragma can be specified either in the compiler command line or at the beginning of the source file before any other text except comments.

Using the New Definition for `errno`

The C-series compiler defines `errno` as a variable with negative values. The D-series compiler defines `errno` as a macro with positive values. You might need to recompile your program using the new `errno` definitions in the `errnoh` header file. You must modify and recompile your program if:

- ☐ It explicitly examines `errno` values or performs a general check for negative `errno` values.
- ☐ It uses literal values for `errno` that have changed.
- ☐ It declares `errno` as:

  ```
  extern int errno
  ```

  Replace this declaration with:

  ```
  #include <errno.h>
  ```

- ☐ It has the potential to export `errno` values to another process that expects to receive the old values (you can remap the new values to the old values).

Result of the `sizeof` Operator

Under the wide-data model, the `sizeof` operator returns a different value for constant operands. On C-series systems, `sizeof` returns 2 or 4. On D-series systems, `sizeof` always returns 4. This change was made for ANSI compliance. You must change programs that rely on the `sizeof` operator returning a 2 for constant operands.

Type of `size_t`

`size_t`, the result of the `sizeof` operator, has changed from type long to type unsigned long for programs that use the large-memory model or the wide-data model. You must change programs that rely on `size_t` being of type long.

`fflush` Function

On C-series systems, the `fflush` function can flush partial lines to `stdout` (standard output) or `stderr` (standard error) file if the file is redirected to type 101 edit files. On D-series systems, the `fflush` function cannot flush partial lines to `stdout` and `stderr` because these files are shared resources under the Common Run-Time Environment, the CRE.

If your program must flush partial lines to `stdout` or `stderr`, you can either open the file using the `freopen` function or close and reopen the file using the `fopen` function. In both cases, the file cannot be shared.

`sscanf` Function

The sscanf function returns a value that is the number of items scanned and converted. On C-series systems, the `sscanf` function returns a -1 if no conversion takes place. On D-series systems, the `sscanf` function returns a 0 if no conversion takes place. You must change your programs if they rely on `sscanf` returning a -1.

Changing Keywords

You should change the following keywords in your program:

C-Series Keyword (Superseded)	D-Series Keyword
`cc_status`	`_cc_status`
`lowmem`	`_lowmem`
`extensible`	`_extensible`
`variable`	`_variable`
`tal`	`_tal`

Replacing Obsolete TAL Function Declarations

You should declare TAL functions using a format like that used in the following example:

```
_tal long tal_function (extptr char *p, short i);
```

Declaring Function Prototypes

The C-series compiler generates an implicit function prototype at the call to a function that does not have a function prototype specified in the scope of the call. This implicit prototyping violates the ANSI standard and results in code that is not portable.

The D-series compiler does not generate these prototypes. You should include a function prototype within the scope of each call to a function. If you do not include a function prototype, the compiler promotes parameters at each call site according to the ANSI standard rules.

Program Elements Affected by D-Series System Enhancements

D-Series system changes that enhance system performance can affect a C-series program running on a D-series system. You may need to modify the following program elements:

- ☐ CPU and PIN variables
- ☐ File-system error numbers
- ☐ File names, including disk file names, device names, and process file names
- ☐ Process identifiers, including process IDs, process handles, and process descriptors
- ☐ Subvolume defaulting in disk file names
- ☐ Guardian procedure calls

Declaring CPU and PIN Variables

Declare all PIN values, including backup-process PIN values, as `short` variables. Declare all CPU values as a separate `short` variables.

For example:

C-Series Declarations (Superseded)	D-Series Declarations
`short cpu_pin;` `char pin;`	`short cpu, pin;`

Declaring and Checking File-System Error Numbers

A D-series file-system error number can be a maximum of 16 bits. If you call Guardian procedures, you might need to convert the parts of your program that declare and check file-system errors.

To accommodate the expanded format, declare a file-system error number as a `short` variable:

For example:

C-Series Declaration (Superseded)	D-Series Declaration
`char fs_error_number;`	`short fs_error_number;`

Your program might also include code that sets a maximum value for a file-system error number (for example, 255). Therefore, make sure that your code does not exclude any new error numbers. Also, because Tandem might define additional error numbers in future releases, do not consider currently undefined numbers as invalid.

For a list and description of all file-system error numbers, refer to the *Guardian Procedure Errors and Messages Manual*.

Using Guardian File Names

Guardian file names include:

- ☐ Disk file names
- ☐ Device names (such as a printer or terminal name)
- ☐ Process file names

You might need to convert the parts of your program that declare and use file-name variables.

Disk File Names. Your existing program might declare a Guardian disk-file-name variable. The largest D-series disk file names are:

For permanent files	35 bytes (one byte larger than the external form of a C-series network file name)
For temporary files	26 bytes (4 bytes larger than the external form of a C-series network file name)

When accessing Guardian disk files on remote D-series systems in a network, a converted program can use a network disk file name with an eight-character volume name (one to seven characters after the dollar sign). A C-series network disk file name allows a maximum of six characters after the dollar sign in the volume name.

Therefore, you might need to declare your network file-name variables large enough to include this extra character and the terminating null character. To ensure that your declaration is long enough, use the ZSYS_VAL_LEN_FILENAME constant from the ZSYSC file and add one for the null character. For example:

```
char employee[ZSYS_VAL_LEN_FILENAME + 1] =
                        "\\newyork.$payroll.july1990.employee";
char managers[ZSYS_VAL_LEN_FILENAME + 1] =
                         "\\newyork.$disk4.level1.managers";
```

Device Names. Your existing program might declare a variable for a Guardian device name. The largest D-series device names are:

Device name without a node name or qualifier	8 characters
Device name without a node name but with a qualifier	17 characters
Network device name without a qualifier	17 characters
Network device name with a qualifier	26 characters

When accessing devices on remote D-series systems in a network, a converted program can use eight-character network device names (one to seven characters after the dollar sign). A C-series network device name allows a maximum of six characters after the dollar sign.

Therefore, you might need to declare your network device names large enough to include this extra character and the terminating null character. If you use the ZSYSC file, use the ZSYS_VAL_LEN_FILENAME constant and add one for the null character. For example:

```
/* Network device name without a qualifier */

char device_name[19] = "\\hamburg.$term001";

/* Network device name with a qualifier */

char network_device_name[ZSYS_VAL_LEN_FILENAME + 1] =
                "\\hamburg.$lineptr.#room025";
```

Process File Names. Your existing program might declare a variable for a C-series process file name. The D-series operating system uses D-series process file names instead of C-series process file names. Use C-series process file names for compatibility with unconverted C-series procedures.

D-series process files names are variable-length string data items with their lengths specified as separate data items. The following example uses declarations from the ZSYSC file to declare a process file name:

```
/* process file name: */
char process_filename[ZSYS_VAL_LEN_FILENAME + 1];
```

The process-file-name declaration has an extra byte for the terminating null character.

Declaring Process Identifiers

Your existing program might declare a process-ID variable to identify a process (for example, an opener in an opener table):

```
short process_id[4];
```

Convert the process-ID variable declaration to a process-handle variable for process-control operations or to a process-descriptor variable for returning information from a Guardian procedure. Use a process-ID variable for compatibility with unconverted C-series procedures.

A process handle is a 20-byte fixed-length structure. A process descriptor is a specific form of process file name that always includes a node name and sequence number. The following examples use declarations from the ZSYSC file to declare a process handle and a process descriptor:

```
/* process handle: */
short process_handle[ZSYS_VAL_PHANDLE_WLEN];

/* process descriptor: */
char process_descriptor[ZSYS_VAL_LEN_PROCESSDESCR + 1];
```

The process-descriptor declaration has an extra byte for the terminating null character.

Avoiding Subvolume Defaulting in Disk File Names

Your existing program might use subvolume defaulting to represent a Guardian disk file name in the form *volume.file-id*. For example:

```
char disk_file[17] = "$diskvol.filename";
```

Avoid subvolume defaulting in your program. If a file name requires the volume name, also include the subvolume name:

```
char disk_file[ZSYS_VAL_LEN_FILENAME + 1] =
                                "$diskvol.subvol.filename";
```

Converting Guardian Procedure Calls

Guardian procedure calls that you might need to convert are procedure calls that accept or return:

- ☐ A PIN parameter for a high-PIN process
- ☐ A process-ID parameter

D-series procedures use a process handle (which includes the CPU and PIN values) rather than a process ID to identify a process.

If you convert your program to run at a high PIN, then you must replace MYPID procedure calls with calls to the PROCESS_GETINFO_ and PROCESSHANDLE_DECOMPOSE_ procedures. “Converting a C Program to Run at a High PIN,” later in this section, describes how to do this.

Appendix A lists the D-series procedures that supersede C-series procedures. For a description of each procedure, refer to the *Guardian Procedure Calls Reference Manual*.

For examples of Guardian procedure calls, refer to Section 3, "Converting TAL Applications." If you are converting a C program, you must convert the TAL procedure calls shown in Section 3 to C. The *Common Run-Time Environment (CRE) Programmer's Guide* provides information about converting TAL procedure calls to C.

Note If you are converting an existing C program (or writing a new program), Tandem recommends that you use Guardian procedures rather than ANSI or supplementary I/O C functions for:

- ☐ Performing interprocess communication
- ☐ Opening $RECEIVE and reading or replying to system messages

Making the C Compiler Run as a High-PIN Process

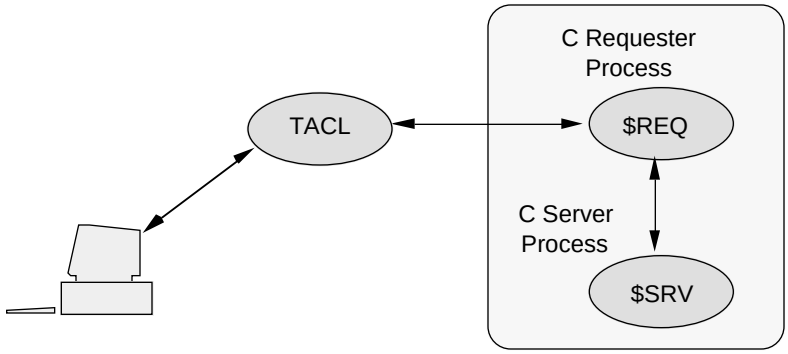
When you start the C compiler using the TACL RUN command, TACL calls the PROCESS_CREATE_ procedure to create the C compiler process and the resulting BINSERV and SYMSERV processes at low PINs.

To run the C compiler process (and the BINSERV and SYMSERV processes) at a high PIN, you must use the Binder program to set the HIGHPIN object-file attribute to ON in the C compiler object file (provided you have the proper authority to change this file). TACL then runs the C compiler (and the BINSERV and SYMSERV processes) at a high PIN if one is available. For more information about TACL, refer to the *TACL Reference Manual.*

Converting a C Program to Run at a High PIN

This subsection describes how to convert your C program to run at a high PIN under the D-series operating system. Figure 5-2 shows a typical application. The box shows which processes this part of the conversion applies to. Converting a C program to run at a high PIN applies to both of these processes.

Figure 5-2. Converting a C Program to Run at a High PIN



To convert your program, you must:

- ☐ Set the HIGHPIN object-file attribute (which tells the system that your program can run at a high PIN)
- ☐ Make sure that each library file your program uses also has its HIGHPIN object-file attribute set (and is capable of running at a high PIN)
- ☐ Declare PIN variables large enough to hold high-PIN values
- ☐ Convert all MYPID Guardian procedure calls

These topics are described in the following subsections.

Setting the HIGHPIN Object-File Attribute

The HIGHPIN object-file attribute directs the system to run a program at a high PIN if one is available. If a high PIN is not available, the program runs at a low PIN if one is available. You set the HIGHPIN object-file attribute either during compilation using a compiler pragma or after compilation using the Binder program.

To set the attribute when you compile your program, specify the HIGHPIN pragma in your source code or as a compiler option in the TACL RUN command for the C compiler. The BINSERV program then sets the HIGHPIN attribute in the object file. An example of this pragma in a source file is:

```
#pragma HIGHPIN
```

An example of this pragma as a compiler option is:

```
10> C /IN csrc, OUT $s.#clst, NOWAIT/ cobj; HIGHPIN
```

You need to specify the HIGHPIN pragma only once during a compilation. However, you can specify it any number of times and the compiler will not generate an error.

If you do not set the HIGHPIN attribute when you compile your program, you can set it after compilation using Binder. For a single object file, use the Binder CHANGE command:

```
@CHANGE HIGHPIN ON IN cobj
```

If you are binding more than one object file into a single target object file, use the Binder SET command to set the HIGHPIN object-file attribute in the target file. For the target object file to have its HIGHPIN object-file attribute set, each constituent object file must also have its HIGHPIN attribute set.

Using a Library File

If your existing program uses a library file, the library file must also have its HIGHPIN object-file attribute set. To determine the current setting of the HIGHPIN attribute for a library file, use the Binder SHOW command:

```
@SHOW SET HIGHPIN FROM libfile
```

If necessary, set this attribute as described in the previous subsection (provided the library file has been converted to support a high-PIN process).

Declaring CPU and PIN Variables

As stated earlier under “Declaring and Using Guardian Variables,” your existing program might declare a `short` variable for both the CPU and PIN values or a `char` variable for a PIN value:

```
short cpu_pin;
char pin;
```

Declare all PIN values, including backup-process PIN values, as `short` variables. Declare a CPU value as a separate `short` variable:

```
short cpu, pin;
```

Converting MYPID Procedure Calls

Your existing program might call the MYPID procedure to obtain its CPU and PIN values:

```
short cpu_pin;

...

cpu_pin = MYPID();
```

If a high-PIN process calls MYPID, a trap condition occurs. You must convert all MYPID calls into calls to the PROCESSHANDLE_DECOMPOSE_ procedure.

The PROCESSHANDLE_DECOMPOSE_ procedure requires a process handle as an input parameter. If you do not know the process handle of your process, first call the PROCESSHANDLE_GETMINE_ procedure. Then pass the result to PROCESSHANDLE_DECOMPOSE_ , which returns the CPU and PIN values as separate integer values. For example:

```
short my_cpu, my_pin;
...

/* Return my process handle */
status = PROCESSHANDLE_GETMINE_(my_phandle);

if (status != 0) err_routine (status);

/* Return my CPU and PIN values */
status = PROCESSHANDLE_DECOMPOSE_ (my_phandle,
                                   my_cpu,
                                   my_pin);
if (status != 0) err_routine (status);
```

Your existing program might also call the MYPID procedure within another Guardian procedure call (for example, in a SETMODE function 11, GETCRTPID, or PROCESSINFO call). This example shows MYPID in a SETMODE (function 11) procedure call:

```
status = SETMODE (file_number,
                  11,
                  MYPID(),   /* return cpu and pin */
                  0,
                  previous_owner);
```

For SETMODE function 11, you are not required to set the *parameter_1* value to the CPU and PIN values. Instead, set *parameter_1* to any positive value:

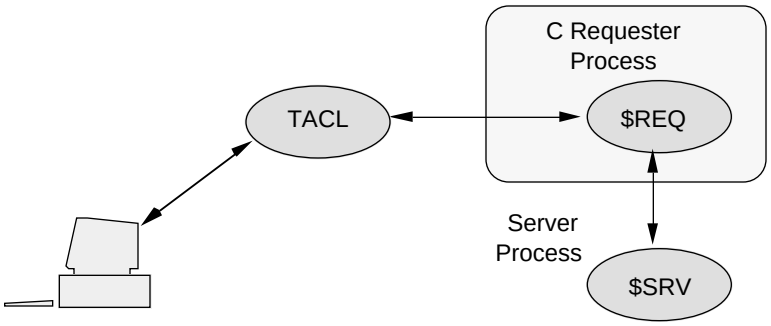
```
...
status = SETMODE (file_number,
                  11,
                  1,  /* set to any positive value */
                  0,
                  previous_owner);
```

Creating a High-PIN Process

You can create a high-PIN process programmatically or interactively. This subsection describes how to programmatically create a high-PIN process. For information on how to interactively create a high-PIN process using TACL, see Section 7, "Converting TACL Programs."

Figure 5-3 shows the processes involved in converting a typical application. The steps described in this subsection apply to the requester process $REQ, which creates the high-PIN server process $SRV.

Figure 5-3. Converting a C Program to Create a High-PIN Process



Your existing program might create a new process using the NEWPROCESS[NOWAIT] procedure:

```
#include <cextdecs(OPEN,                    \
                   NEWPROCESS,              \
                   WRITEREADX,              \
                   CLOSE)> nolist
#include "$system.zsysdefs.zsysc"
...

lowmem short program_file[12];
lowmem short home_terminal[12];
lowmem short process_id[4];
lowmem short process_name[3];
short err_return, priority, memory_pages, cpu;
short error_info[2];
...
```

```
NEWPROCESS (program_file,
            priority,
            memory_pages,
            cpu,
            process_id,
            &err_return,
            process_name,
            home_terminal,
            /* flags  */ 0,
            /* job_id */  ,
            error_info);
```

On a D-series system, the NEWPROCESS[NOWAIT] procedure can create only a low-PIN process. The D-series operating system provides the PROCESS_CREATE_ procedure to create a new low-PIN or high-PIN process in a waited or nowait manner.

The following procedure creates a new process in a waited manner. The system returns the results in the returned value *error* and *error_detail* parameter:

```
#include <cextdecs(FILE_OPEN_,              \
                   PROCESS_CREATE_ ,        \
                   WRITEREADX,              \
                   FILE_CLOSE_)> nolist
...

char program_file[ZSYS_VAL_LEN_FILENAME + 1];
short process_handle[ZSYS_VAL_PHANDLE_WLEN];

short pf_length;
short error, priority, memory_pages, cpu, error_detail;

...
error = PROCESS_CREATE_(program_file, pf_length,
                        /* library_file   */ ,
                        /* lf_length      */ ,
                        /* swapfile       */ ,
                        /* sf_length      */ ,
                        /* ext_swapfile   */ ,
                        /* esf_length     */ ,
                        priority,
                        cpu,
                        process_handle,
                        error_detail);
```

If your program creates a new process in a nowait manner, the system returns the results in system message -102 (PROCESS_CREATE_ completion), which is analogous to system message -12 (NEWPROCESSNOWAIT completion). You read system message -102 from $RECEIVE using the READ[X] or READUPDATE[X] procedure.

For additional information on creating processes in a nowait manner, creating processes at a low PIN, and other information about the PROCESS_CREATE_ procedure, refer to Section 3, “Converting TAL Applications.”

Opening and Communicating With a High-PIN Server

Your existing program might be a requester that communicates with a server. For example, you might open a server, send it a request, and then process its reply. You might also open a server for a backup requester if your program is running as a process pair.

The degree of conversion you need to perform depends on whether your server is named or unnamed, and, if the server is named, on how long the name is. Your options are as follows:

- ☐ If the server is local and named or if the server is remote with a name of five characters or less (including the dollar sign), then no conversion is necessary. You can still open the high-PIN server using the Guardian C-series-compatible OPEN procedure. See Appendix C, “System Compatibility,” for further information on communicating with a named high-PIN process.
- ☐ If the server is remote and has a six-character name, then you need to first convert your requester to run at a high PIN as described under “Converting a C Program to Run at a High PIN” earlier in this section, and then complete the conversion as described under “Communicating With a High-PIN Server” and “Monitoring a High-PIN Server,” later in this section. See Appendix C, “System Compatibility,” for further information on communicating with a named high-PIN process.
- ☐ If the server is unnamed, then you have the following options:
  - ☐ Set the RUNNAMED object-file attribute in the server so that the system provides a name for the server, and pass the system-assigned name to the requester; for example, in a DEFINE or an ASSIGN. See “Setting the RUNNAMED Object-File Attribute” later in this section for details.
  - ☐ Convert the requester to run at a high PIN as described under “Converting a C Program to Run at a High PIN” earlier in this section, and then complete the conversion as described under “Communicating With a High-PIN Server” and “Monitoring a High-PIN Server,” later in this section.

For information about converting a server to monitor a high-PIN requester process, including maintaining an opener table, refer to “Being Opened by and Communicating With a High-PIN Requester,” later in this section.

Setting the RUNNAMED Object-File Attribute

The RUNNAMED object-file attribute causes a process to run as a named process even if you do not provide a name for it. Thus, a process can run at a high PIN under the D-series operating system and be opened by an unconverted process using the Guardian OPEN procedure. For more information about how an unconverted process running on a D-series system can communicate with a named high-PIN process, refer to Appendix C, “System Compatibility.”

You set the RUNNAMED object-file attribute either during compilation using a compiler pragma or after compilation using the Binder program.

To set the attribute when you compile your program, specify the RUNNAMED pragma in your source code or as a compiler option in the TACL RUN command for the C compiler. The BINSERV program then sets the RUNNAMED attribute in the object file. An example of this pragma (with the HIGHPIN pragma) in a source file is:

```
#pragma HIGHPIN, RUNNAMED
```

An example of this pragma as a compiler option is:

```
10> C /IN csrc, OUT $s.#clst, NOWAIT/ cobj; HIGHPIN, RUNNAMED
```

You need to specify the RUNNAMED pragma only once during a compilation. However, you can specify it any number of times and the compiler will not generate an error.

If you do not set the RUNNAMED attribute when you compile your program, you can set it after compilation using Binder. For a single object file, use the Binder CHANGE command:

```
@CHANGE RUNNAMED ON IN cobj
```

If you are binding more than one object file into a single target object file, use the Binder SET command to set the RUNNAMED object-file attribute. If any of the constituent object files used to build the target file has the RUNNAMED object-file attribute set, Binder sets this attribute in the target object file.

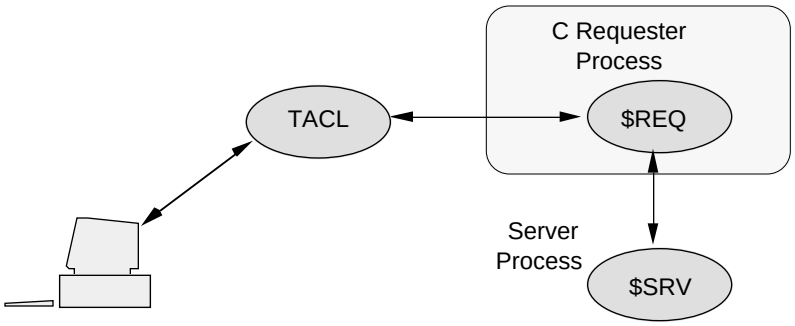
Communicating With a High-PIN Server

A requester can open and communicate with a high-PIN named server by opening the server using the OPEN procedure. However, you must convert your requester to open the server using the FILE_OPEN_ procedure if the server:

- Is unnamed
- Is on a remote D-series system and has a six-character name (a dollar sign and five alphanumeric characters)

Figure 5-4 shows the processes involved in converting this part of a typical application. The steps in this subsection apply to the requester process $REQ.

Figure 5-4. Converting a C Requester to Communicate With a High-PIN Server



This subsection discusses converting the following operations:

- Opening and closing the high-PIN server
- Opening and closing the high-PIN server for a backup process
- Sending requests to the high-PIN server

Opening a High-PIN Server

Your requester might open the server using the OPEN procedure:

```
short server_name[12] = "$SRV                        ";

...

c_code = OPEN (server_name,
               server_file_number,
               nowait_depth,
               sync_depth);
```

Convert your requester to open the high-PIN server using the FILE_OPEN_ procedure. The FILE_OPEN_ procedure requires a variable-length string for the server file-name input parameter rather than the 12-word internal-format file name.

Note If the *file-name* input parameter is incomplete (that is, not fully qualified), FILE_OPEN_ uses the current settings, including the system name, in the =_DEFAULTS DEFINE for the unspecified parts.

FILE_OPEN_ also accepts a DEFINE name that represents a valid file name in this format.

FILE_OPEN_ accepts an integer *options* parameter to specify certain file characteristics. The *options* bit positions represent these options:

options Bit Position	Description
0	Allow unstructured access for a disk file (must be 0 for other files and devices)
1	Execute a nowait open
2	Do not execute an update when the file is opened
3	Use any available file number for backup open (0 means use the same file number as in the primary open)
4 through 13	Reserved; must be 0
14	Receive C-series system messages ($RECEIVE only)
15	Do not receive process open and close system messages ($RECEIVE only)

The ZSYSC file contains constant declarations that you can use with the *options* parameter.

If you started the server using the PROCESS_CREATE_ procedure, you can use the PROCESS_CREATE_ process-descriptor output parameter directly in the FILE_OPEN_ procedure call (shown below as the *server_name* parameter). Refer to "Creating and Managing a High-PIN Process" earlier in this section for details.

```
error = FILE_OPEN_(server_name,
                   server_length,
                   server_file_number,
                   exclusion_mode,
                   nowait_operations,
                   sync_depth,
                   options);
```

If you open the server using the nowait open option, you must call the AWAITIO[X] procedure to complete the open. To determine the *error* and *options* values, call the FILE_GETINFOLIST_ procedure and check the items specified by ZSYS_VAL_FINF_LASTERROR and ZSYS_VAL_FINF_OPENOPTS, respectively (provided you use the ZSYSC file).

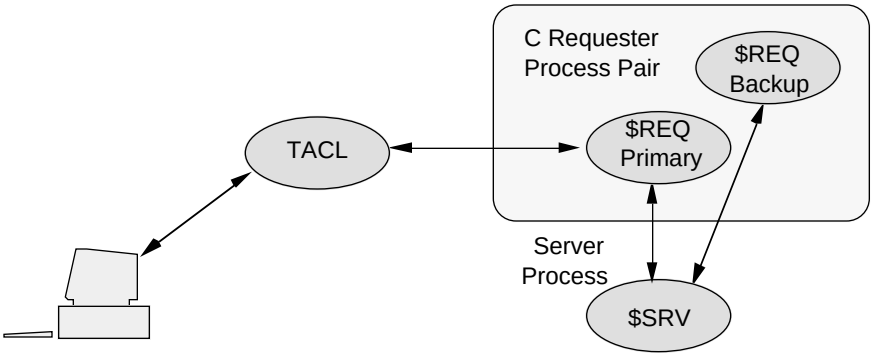
Opening a High-PIN Server for a Backup Requester Process

If your requester is running as a process pair, it might open the server for its backup process using the CHECKOPEN procedure:

```
c_code = CHECKOPEN (server_name,
                    server_file_number,
                    nowait_depth,
                    sync_depth,
                    /* seq_block_buffer */ ,
                    /* buffer_length    */ ,
                    back_error);
```

Figure 5-5 shows a requester process pair and a server process.

Figure 5-5. Opening a High-PIN Server for a Backup Process



Convert your requester to open the high-PIN server for its backup process using the FILE_OPEN_CHKPT_ procedure. To identify the high-PIN server, FILE_OPEN_CHKPT_ requires the file number returned by the FILE_OPEN_ procedure call in the primary process. The system returns a file-system error (if a file-system error occurs) in the returned value *error* and the status of the backup open in an output parameter, which is the *backup_open_status* parameter in this example:

```
error = FILE_OPEN_CHKPT_(server_file_number,
                         backup_open_status);
```

If you opened the server using the nowait open option, you must call the AWAITIO[X] procedure to complete the open. To determine the *error* and *backup_open_status* values, call the FILE_GETINFOLIST_ procedure and check the items specified by ZSYS_VAL_FINF_LASTERROR and ZSYS_VAL_FINF_LASTERRORDETAIL, respectively (provided you use the ZSYSC file).

Sending a Request to a High-PIN Server

Your requester might send a request to a high-PIN server using the WRITE[X] or WRITEREAD[X] procedure:

```
c_code = WRITEREADX (server_file_number,
                     sbuffer,
                     write_count,
                     read_count,
                     count_read);
```

Your WRITE[X] or WRITEREAD[X] procedure call should not require any changes to send a request to a high-PIN server.

Closing a High-PIN Server

Your requester might close the server using the CLOSE procedure:

```
CLOSE (server_file_number);
```

You can close a high-PIN server using either the CLOSE or FILE_CLOSE_ procedure:

```
error = FILE_CLOSE_(server_file_number);
```

Closing a High-PIN Server for a Backup Requester Process

Your requester might close the server for the backup process using the CHECKCLOSE procedure:

```
CHECKCLOSE (server_file_number);
```

You can close the server for the backup process using either the CLOSE procedure or the FILE_CLOSE_CHKPT_ procedure:

```
error = FILE_CLOSE_CHKPT_(server_file_number);
```

Monitoring a High-PIN Server

If your program monitors a high-PIN server, you must convert the following operations:

- ☐ Opening and closing $RECEIVE
- ☐ Reading process-deletion and status-change messages
- ☐ Using the CHILD_LOST_ procedure

The following paragraphs describe how to convert these operations. These steps also can apply to any creator process that monitors a process that it has created.

Opening $RECEIVE

Your requester might open $RECEIVE using the OPEN procedure:

```
short receive_name[12] = "$RECEIVE                ";

...

c_code = OPEN (receive_name,
               receive_file_number,
               read_open_close_msgs,
               receive_depth);
```

Convert your requester to open $RECEIVE using the FILE_OPEN_ procedure. Use a file-name string for the $RECEIVE file name instead of the internal file-name format. Specify the length as a separate integer value.

Bit 14 of the *options* parameter must be zero (which is the default value) for the system to send D-series system messages to $RECEIVE; otherwise, the system sends C-series system messages to $RECEIVE for the requester.

An example of a FILE_OPEN_ procedure call for $RECEIVE is:

```
#define receive_name_length 8;

char receive_name[receive_name_len + 1] = "$RECEIVE";

...

/* Open $RECEIVE to read D-series system messages */

error = FILE_OPEN_(receive_name,
                   receive_name_length,
                   receive_file_number,
                   /* access_mode       */ ,
                   /* exclusion_mode    */ ,
                   /* nowait_operations */ ,
                   receive_depth,
                   options);
```

Reading System Messages From $RECEIVE

Your requester might read system messages from $RECEIVE using the READ[X] or READUPDATE[X] procedure:

```
char message_buffer[200]; /* Message buffer (200 bytes) */

...

read_count = 200;

c_code = READX (receive_file_number,
                message_buffer,
                read_count,
                bytes_read);
```

The lengths shown for each system message are subject to change. In a future release, Tandem might add new fields to the end of a system message (while maintaining the layout of the existing fields). Therefore, use a READ[X] or READUPDATE[X] message buffer at least 250 bytes in length. Also, use a *read_count* parameter of 250 bytes.

If you use the ZSYSC file, use the ZSYS_VAL_SMSG_LEN constant declaration to specify the system message length in bytes. If you work in words you can use the ZSYS_VAL_SMSG_WLEN constant declaration instead.

```
char message_buffer[ZSYS_VAL_SMSG_LEN];

...

read_count = ZSYS_VAL_SMSG_LEN;

c_code = READX (receive_file_number,
                message_buffer,
                read_count,
                bytes_read);
```

The ZSYSC file also contains structures that you can use when your requester reads system messages.

Reading Process-Deletion System Messages. Your requester might monitor a server process by reading these process-deletion system messages from $RECEIVE:

-2 CPU down: named process deletion
-5 Process normal deletion: stop
-6 Process abnormal deletion: abend

Convert your requester to read and process the D-series system message -101 (Process deletion), which supersedes all the above messages.

Reading Status-Change System Messages. Your requester might monitor a server process by reading these status-change system messages from $RECEIVE:

-2 CPU down: local CPU failure after process called MONITORCPUS
-8 Change in status of network node

Continue to read system message -2. Then, convert your requester to read these new status-change messages, all of which supersede system message -8:

-100 Remote CPU down
-110 Loss of communication with node
-113 Remote CPU up

To receive system messages -100, -110, and -113, first call the MONITORNET procedure with the *enable* parameter set to 1.

Processing System Messages Using the CHILD_LOST_ Procedure

Your requester might call a user-written routine to determine whether a process-deletion or status-change message affects the server.

You might convert your requester to call the new CHILD_LOST_ procedure. The CHILD_LOST_ procedure accepts the process handle of a process you are monitoring and either a C-series (-2, -5, -6, or -8) or D-series (-2, -100, -101, -110, or -113) process-deletion or status-change system message:

```
error = CHILD_LOST_(message,
                    message_length,
                    process_handle);
```

The CHILD_LOST_ *error* returned value indicates whether the process (or process pair) is lost:

0 The process (or process pair) is not lost.
4 The process (or process pair) is lost.

Note System message -101 (Process deletion) contains the process handle and process descriptor of the process that terminated. If a named process (or process pair) has terminated, this is the last opportunity for you to save the process name of the process (or process pair).

Closing $RECEIVE

Your requester might close $RECEIVE using the CLOSE procedure:

```
CLOSE (receive_file_number);
```

You can close $RECEIVE using either the CLOSE or FILE_CLOSE_ procedure:

```
error = FILE_CLOSE_(receive_file_number);
```

Being Opened by and Communicating With a High-PIN Requester

This subsection describes how to convert a server process written in C to communicate with a high-PIN requester. Whether you need to convert the server process depends in part on whether the server tracks its openers. If the server does keep track of its openers, you should enable the server to run at a high PIN as described in "Converting a C Program to Run at a High PIN," earlier in this section, and then convert the server as described under "Converting a Server," later in this subsection.

If the server does not track its openers, or if you choose not to perform the conversion, then you can keep the server process at a low PIN and not convert it, except for setting the HIGHREQUESTERS object-file attribute as described under "Setting the HIGHREQUESTERS Attribute to Allow High-PIN Openers," later in this subsection. Setting this attribute enables a high-PIN requester to open a low-PIN server.

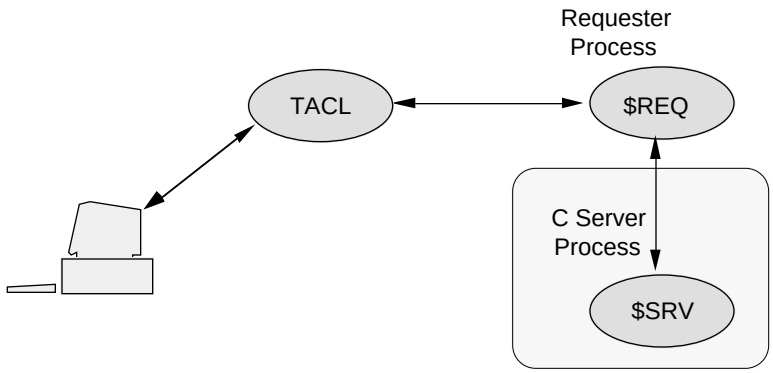
Converting a Server

If your server process tracks its openers, you must convert the following parts of your program:

- ☐ Defining an opener table
- ☐ Opening $RECEIVE
- ☐ Reading D-series system messages from $RECEIVE
- ☐ Getting information about system messages
- ☐ Processing system messages
- ☐ Replying to a system message
- ☐ Using the OPENER_LOST_ procedure to maintain an opener table

Figure 5-6 shows the processes involved in converting an application. The steps described in this subsection apply to the server process $SRV.

Figure 5-6. Converting a C Server to Communicate With a High-PIN Requester



Defining an Opener Table

If your server tracks its openers, it might define an opener table that uses a process ID to identify an opener (primary process opener and backup process opener):

```
struct opener_table
  {
    short current_count;
    struct openers[max_openers]
      {
        short primary_process_id[4];
        short primary_file_number;
        short backup_process_id[4];
        short backup_file_number;
      };
  };
```

Convert your opener table to identify an opener using a process handle rather than a process ID. To use the OPENER_LOST_ procedure (which is described later in this subsection) to manage your opener table, define the table as follows:

- ☐ Use a process handle to identify both a primary-process and backup-process opener.
- ☐ Declare the process-handle field for the backup-process opener immediately after the process-handle field for the primary-process opener (that is, the fields must be stored in a 20-word contiguous part of an entry).
- ☐ Declare table entries as fixed length and contiguous.
- ☐ Do not store variable-length items in the table. If necessary, save a pointer in the table to a variable-length item.
- ☐ Set the process handles for primary and backup openers in unused entries to null values (all -1s).

An example of an opener table that the OPENER_LOST_ procedure can process is:

```
struct opener_table
  {
    short current_count;
    struct openers[max_openers]
      {
        short primary_process_handle[ZSYS_VAL_PHANDLE_WLEN];
        short backup_process_handle[ZSYS_VAL_PHANDLE_WLEN];
        short primary_file_number;
        short backup_file_number;
      };
  };
```

Opening $RECEIVE

Your server might open $RECEIVE using the OPEN procedure with bit 1 of the *flags* parameter set to 1 (*flags* = 040000). This allows you to receive system messages such as -30 (Process open) and -31 (Process close):

```
short receive_name[12] = "$RECEIVE               ";

short read_open_close_msgs = 040000 ;

...

status = OPEN (receive_name,
               receive_file_number,
               read_open_close_msgs, /* Value = octal 40000*/
               receive_depth);
```

Convert your server to open $RECEIVE using the FILE_OPEN_ procedure:

1. Use a file-name string for the $RECEIVE file name instead of the internal file-name format. Specify the length as a separate integer value.
2. Make sure that bit 15 of the FILE_OPEN_ *options* parameter is zero (the default value). If this bit is not zero, system messages such as -103 (Process open) and -104 (Process close) are not sent to $RECEIVE.
3. Make sure that bit 14 of the FILE_OPEN_ *options* parameter is zero (the default value) so that the system sends D-series system messages to $RECEIVE. If this bit is not zero, the system sends C-series system messages to $RECEIVE.
4. Set any other FILE_OPEN_ input parameters as required and call the procedure:

```
#define receive_name_length 8;

char receive_name[receive_name_length + 1] = "$RECEIVE";

...

/* Open $RECEIVE to read D-series system messages */

error = FILE_OPEN_(receive_name,
                   receive_name_length,
                   receive_file_number,
                   /* access_mode        */ ,
                   /* exclusion_mode     */ ,
                   nowait_operations,
                   receive_depth);
```

If you open $RECEIVE using the FILE_OPEN_ procedure, the system assumes that you support high-PIN requesters (provided bit 14 of the *options* parameter is zero). You do not need to explicitly set the HIGHREQUESTERS object-file attribute in your server's object file.

When you close $RECEIVE, use either the CLOSE or FILE_CLOSE_ procedure.

Reading System Messages From $RECEIVE

Your server might read system messages from $RECEIVE using the READ[X] or READUPDATE[X] procedure:

```
char message_buffer[200];  /* Message buffer (200 bytes) */

...

read_count = 200;

c_code = READX (receive_file_number,
                message_buffer,
                read_count,
                bytes_read);
```

The lengths shown for each system message are subject to change. Use a READ[X] or READUPDATE[X] message buffer at least 250 bytes in length. Also, use a *read_count* parameter value of 250 bytes.

If you use the declarations in the ZSYSC file, use the ZSYS_VAL_SMSG_LEN constant for the system-message length in bytes or the ZSYS_VAL_SMSG_WLEN constant for the length in words:

```
char message_buffer[ZSYS_VAL_SMSG_LEN];

...

read_count = ZSYS_VAL_SMSG_LEN;

c_code = READX (receive_file_number,
                message_buffer,
                read_count,
                bytes_read);
```

Getting Information About System Messages

Your server might call the RECEIVEINFO or LASTRECEIVE procedure to obtain information about the last message read from $RECEIVE:

```
RECEIVEINFO (process_id,
             message_tag,
             sync_id,
             file_number,
             read_count,
             io_type);
```

Convert the RECEIVEINFO or LASTRECEIVE call into a call to the FILE_GETRECEIVEINFO_ procedure:

```
/* Return information about the last message */

error = FILE_GETRECEIVEINFO_(message_info);
```

FILE_GETRECEIVEINFO_ returns information in the 17-word *message_info* parameter, which has the format shown in Table 5-1. The ZSYSC file contains a structure that you can use for the *message_info* format.

Table 5-1. FILE_GETRECEIVEINFO_ message_info Parameter Format

Word	Description
0	I/O type for the message: 0 = A system message was sent. 1 = The sender called WRITE[X]. 2 = The sender called READ[X]. 3 = The sender called WRITEREAD[X].
1	The maximum reply count in bytes
2	The message tag identifying the message
3	The file number for the message
4 through 5	The sync ID for the message
6 through 15	The process handle of the process sending the message
16	The *open_label* from a previous reply (or -1 if unavailable or for a C-series message)

Reading and Processing Open and Close System Messages

To monitor an opener, your server might read the C-series -30 (Process open) and -31 (Process close) system messages from $RECEIVE.

To monitor a high-PIN process, convert your server to read the D-series -103 (Process open) and -104 (Process close) system messages. When your server is opened or closed by a process pair, it receives a process-open or process-close message from each process of the pair.

If you call the RECEIVEINFO or LASTRECEIVE procedure to obtain information about the process-open or process-close message, convert the call into a call to the FILE_GETRECEIVEINFO_ procedure as described under “Getting Information About System Messages,” earlier in this section.

After calling FILE_GETRECEIVEINFO_, update your opener table using the process handle rather than the process ID to identify the opener.

System Message -103 (Process Open). Check bit 15 of *sysmsg*[7] of the process open message (or `zsys_ddl_smsg_open.z_flags` if you use the ZSYSC file), which indicates whether the opener is a primary or backup process:

- ☐ Primary open (*sysmsg*[7] bit 15 is 0): Add an entry in your opener table for the process.
- ☐ Backup open (*sysmsg*[7] bit 15 is 1): Process a backup open as follows:

  1. Get the process handle for the primary opener from the process-open system message (-103). This process handle is in *sysmsg*[8] for ten words (or the `zsys_ddl_smsg_open.z_primary` field if you use the ZSYSC file).
  2. Use the process handle to search your opener table for the corresponding primary-process open entry. If you find this entry but there is no backup open yet (the backup process handle is null), add the backup process handle to the table entry.
  3. If the primary-process open entry is not found, reject the backup open with a file-system error greater than 9.

System Message -104 (Process Close). Delete the opener-table entry for this process. You should receive a process-close message from each process of a process pair.

Reading and Processing Status-Change Messages

If one of your openers has a CPU failure, or if its system fails or becomes partitioned from your system because of a network failure, you do not receive a process-close message (-31). Therefore, when maintaining an opener table, your server might read and process these status-change messages:

-2 CPU down: local CPU failure after the process called MONITORCPUS
-8 Change in status of network node

Continue to read system message -2. In addition, read these new status-change messages (all of which supersede C-series system message -8):

-100 Remote CPU down
-110 Loss of communication with node
-113 Remote CPU up

To receive system messages -100, -110, and -113, first call the MONITORNET procedure with the *enable* parameter set to 1.

Replying to a System Message

Your server might reply to a system message using the Guardian REPLY[X] procedure:

```
status = REPLYX (reply_buffer,
                 write_count,
                 count_written,
                 message_tag,
                 error_return);
```

Replying to System Message -103 (Process Open). The D-series system supports returning a label value in the reply to a system message -103 (Process open). Typically, an opener-table index gets sent in this way. This label then appears in the *open_label* field of future FILE_GETRECEIVEINFO_ procedure calls that provide information about messages received from the same requester. To support this feature, the file system expects a reply buffer with a length of 0 to 4 bytes; otherwise, the open in the requester returns an error.

Your server might reply to an Open message as follows:

```
write_count = any_valid_integer;
status = REPLYX(reply_buffer,
                write_count,
                /* count_written */ ,
                /* message_tag   */ ,
                error_return);
```

To make use of the *open_label* field in the FILE_GETRECEIVEINFO_ procedure, you must convert your code to reply to the Open message as follows:

```
reply_buffer[0] = -103;
reply_buffer[1] = open_label_value;
write_count = 4;
status = REPLYX (reply_buffer,
                 write_count,
                 /* count_written */ ,
                 /* message_tag   */ ,
                 error_return);
```

If you do not want to use the *open_label* field, you still need to be sure that the reply buffer has a length of 0 to 4 bytes. Convert your server as follows:

```
write_count = 0;
status = REPLYX (reply_buffer,
                 write_count,
                 /* count_written */ ,
                 /* message_tag   */ ,
                 error_return);
```

Replying to an Unknown System Message. Your server should be able to handle an unknown system message. If the first word of a message contains an unknown message number, call the REPLY[X] procedure with an error indication of 2 (invalid operation):

```
status = REPLYX (/* reply_buffer  */ ,
                 /* write_count   */ ,
                 /* count_written */ ,
                 /* message_tag   */ ,
                 invalid_operation);   /* Value is 2 */
```

Using the OPENER_LOST_ Procedure to Maintain an Opener Table

After receiving a status-change message, your server might call one or more routines to maintain its opener table.

You might want to use the OPENER_LOST_ procedure to maintain your opener table. OPENER_LOST_ determines whether a status-change message affects your opener table and updates the appropriate table entry if an opener was lost.

OPENER_LOST_ accepts a C-series (-2 or -8) or D-series (-2, -100, -110, or -113) status-change message and searches your opener table for any processes affected by the message. If OPENER_LOST_ determines that an opener has been lost, it updates the opener-table entry and returns the index of the entry and an *error* value. The *error* value indicates the reason for the opener-table change:

error Value	Reason
4	A backup process opener is lost
5	A primary process opener is lost; the backup process is now the primary process
6	The primary process and backup process (if it exists) openers for a table entry are lost; the table entry is now free

When OPENER_LOST_ returns an *error* value of zero, processing is complete for the message.

To process all entries in your opener table for a status-change message, set up a loop similar to the one shown below. The opener table must be defined as described under "Defining an Opener Table," earlier in this subsection.

```
done  =  0;    /* Set control for start of loop */
index = -1;    /* Set index for start of loop */

do
   {
   error = OPENER_LOST_(message,message_length,
                        opener_table.openers,
                        index,
                        opener_table.current_count,
                        $LEN(opener_table.openers));

  switch (error)
  {
     case 4 :  /* Processing for lost backup opener */
     case 5 :  /* Processing for lost primary opener */
     case 6 :  /* Processing for lost opener */
               /*  (primary and backup for a process pair) */
     default : done = -1   /* Processing is finished or */
                           /*  error occurred */
  };
}
while (done == 0);
```

Setting the HIGHREQUESTERS Attribute to Allow High-PIN Openers

The HIGHREQUESTERS object-file attribute allows a process to support requests from high-PIN requesters. Use the HIGHREQUESTERS object-file attribute only for a source program that contains the main function. You can set the HIGHREQUESTERS object-file attribute by including a compiler pragma in your source file, or you can set it after you have finished converting your source code either using a compiler option or after compilation using the Binder program.

To set the attribute when you compile your program, specify the HIGHREQUESTERS pragma in your source code or as a compiler option in the TACL RUN command for the C compiler. The BINSERV program then sets the HIGHREQUESTERS attribute in the object file. An example of this pragma in a source file is:

```
#pragma HIGHREQUESTERS
```

An example of this pragma as a compiler option is:

```
10> C /IN csrc, OUT $s.#clst, NOWAIT/ cobj; HIGHREQUESTERS
```

You need to specify the HIGHREQUESTERS pragma only once during a compilation. If your existing program includes source code from another file, specify the HIGHREQUESTERS pragma only in the program file that contains the main function; do not specify the pragma in the included file (or files).

If you do not set the HIGHREQUESTERS attribute when you compile your program, you can set it after compilation using Binder. For a single object file, use the Binder CHANGE command:

```
@CHANGE HIGHREQUESTERS ON IN cobj
```

If you are binding more than one object file into a single target object file, use the Binder SET command to set the HIGHREQUESTERS object-file attribute. For Binder to set the HIGHREQUESTERS object-file attribute in a target object file, the object file containing the main function must have this object-file attribute set.

For more information about the HIGHREQUESTERS object-file attribute, refer to “Allowing Opens by High-PIN Requesters” in Appendix C, “System Compatibility.”

6 Converting Pascal Applications

A Pascal program can run at a low PIN under the D-series operating system without any changes. However, for a Pascal program to use the extended features of the D-series operating system, specific parts of it must be converted.

The topics in this section are:

- ☐ Converting basic elements of a Pascal program, such as using the PEXTDECS, PASEXT, and ZSYSPAS files, declaring variables, calling Guardian procedures, and running the Pascal compiler
- ☐ Converting a Pascal program to run at a high PIN
- ☐ Converting a Pascal program to create a high-PIN process
- ☐ Converting a requester to open and communicate with a high-PIN server
- ☐ Converting a server to be opened by and communicate with a high-PIN requester

Section 8, “Converting Other Parts of an Application,” contains information about converting other parts of a Pascal application. For additional information about the Tandem implementation of Pascal, refer to the *Pascal Reference Manual*.

Converting Basic Elements of a Pascal Program

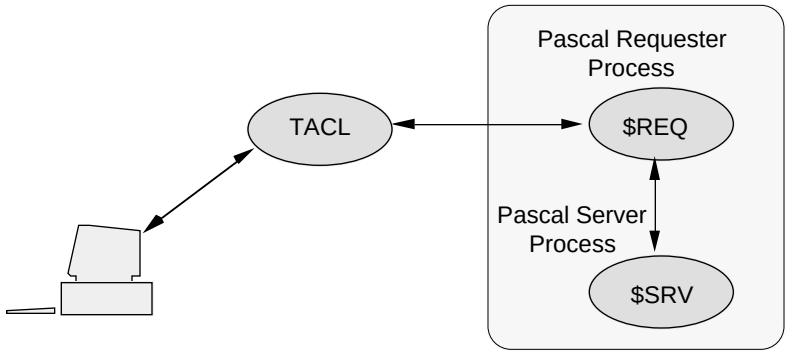
This subsection describes conversion that applies to all Pascal programs you need to convert to run under the D-series operating system, irrespective of what the program does. Later subsections describe how to convert specific functions of your programs such as communicating with a high-PIN process.

This subsection discusses the following topics:

- ☐ Importing source declarations from the PEXTDECS and PASEXT files
- ☐ Importing source declarations from the ZSYSPAS file
- ☐ Declaring and using variables for high PINs, file-system error numbers, file names, and process identifiers
- ☐ Calling Guardian procedures
- ☐ Running the Pascal compiler
- ☐ Binding the Run-time library

Figure 6-1 shows a typical application. The box shows which processes this part of the conversion applies to. Converting basic elements of a Pascal program applies to both of these processes.

Figure 6-1. Converting Basic Elements of a Pascal Program



Importing the PEXTDECS and PASEXT Declarations

The PEXTDECS file contains declarations for Guardian procedures. The PASEXT file contains declarations for Tandem routines. Your existing program should use the SOURCE or CSOURCE compiler directive to import the declarations you need from these files. For example, these SOURCE directives import C-series procedures and routines:

```
IMPORT BEGIN
?SOURCE $SYSTEM.SYSTEM.PEXTDECS (TYPES,
?                                Guardian_OPEN,
?                                Guardian_READ,
?                                Guardian_WRITE,
?                                Guardian_CLOSE)
END; { IMPORT }

IMPORT BEGIN
?SOURCE $SYSTEM.SYSTEM.PASEXT (SYSTEM)
END; { IMPORT }
```

Convert your SOURCE or CSOURCE directive to specify declarations from D-series PEXTDECS or PASEXT files. Use the new procedure names for any D-series procedures. Specify only the D-series PEXTDECS file; the C-series declarations are also available in this file. For example:

```
IMPORT BEGIN
?SOURCE $SYSTEM.SYSTEM.PEXTDECS (TYPES,
?                                FILE_OPEN_,
?                                Guardian_READ,
?                                Guardian_WRITE,
?                                FILE_CLOSE_)
END; { IMPORT }

IMPORT BEGIN
?SOURCE $SYSTEM.SYSTEM.PASEXT (SYSTEM)
END; { IMPORT }
```

Importing the ZSYSPAS Declarations

Tandem provides source declarations for Guardian procedures and system messages in the ZSYSPAS file. This file is typically found on the $SYSTEM.ZSYSDEFS subvolume. Contact your system manager to find the location of this file on your system.

To use these declarations, import them using the SOURCE or CSOURCE compiler directive. For example, this SOURCE directive imports the entire ZSYSPAS file without printing the contents of the file:

```
IMPORT BEGIN
?SOURCE $SYSTEM.ZSYSDEFS.ZSYSPAS, NOLIST
END; { IMPORT }
```

The ZSYSPAS file is divided into sections, which allows you to import only the sections your program actually needs. For example, this SOURCE directive imports only the process-creation and system-message constant declarations and prints the contents of each section:

```
IMPORT BEGIN
?SOURCE $SYSTEM.ZSYSDEFS.ZSYSPAS (PROCESS_CONSTANT,
?                                 SYSTEM_MESSAGES_CONSTANT)
END; { IMPORT }
```

To print a listing of the ZSYSPAS file to check the declarations that are available for your program, use the FUP COPY command:

```
10> FUP COPY $SYSTEM.ZSYSDEFS.ZSYSPAS, $s.#lineptr
```

Naming Standard Files in the Module Heading

The D-series version of the Pascal compiler requires that you name the INPUT, OUTPUT, and STDERR files in a module heading if the module accesses these files using Pascal I/O routines and if the program does not contain a Pascal program heading (that is, if the main program is written in a language other than Pascal). Such a module should have a heading like the following:

```
MODULE chart INPUT, OUTPUT, STDERR
```

Declaring and Using Programming Variables

For your existing program to run at a high PIN, you might need to add or modify declarations for the following variables:

- ☐ CPU and PIN variables
- ☐ File-system error numbers
- ☐ Guardian file names, including disk file names, device names, and process file names
- ☐ Process identifiers, including process IDs, process handles, and process descriptors

Declaring CPU and PIN Variables

Your existing program might declare a 16-bit variable for both the CPU and PIN values:

```
VAR cpu_pin     : INTEGER;
```

Or your program might declare an 8-bit variable for a PIN value:

```
VAR pin         : BYTE;
```

Declare all PIN values, including backup-process PIN values, as 16-bit variables. Declare a CPU value as a separate 16-bit variable:

```
VAR cpu, pin    : INTEGER;
```

Declaring and Checking File-System Error Numbers

A Guardian procedure can return a file-system error number to report an error or special condition. You might need to convert the parts of your program that declare and check file-system errors. For example, your program might declare an 8-bit variable for a file-system error number:

```
VAR fs_error_number  : BYTE;
```

Declare a file-system error number as a 16-bit variable:

```
VAR fs_error_number  : INTEGER;
```

Your program might also include code that sets a maximum value for a file-system error number (for example, 255). A D-series file-system error number can be a maximum of 16 bits. Therefore, make sure that your code does not exclude any new error numbers. Also, because Tandem might define additional error numbers in future releases, do not consider currently undefined numbers as invalid.

For a list and description of all file-system error numbers, refer to the *Guardian Procedure Errors and Messages Manual*.

Using Guardian File Names

Guardian file names include disk file names, device names (such as a printer or terminal name), and process file names. You might need to convert the parts of your program that declare and use file-name variables.

Disk File Names. Your existing program might declare a Guardian disk-file-name variable. The largest D-series disk file names are:

For permanent files	35 bytes (one byte larger than the external form of a C-series network file name)
For temporary files	26 bytes (4 bytes larger than the external form of a C-series network file name)

When accessing Guardian disk files on remote D-series systems in a network, a converted program can use a D-series network disk file name with an eight-character volume name (one to seven characters after the dollar sign). A C-series network disk file name allows a maximum of six characters after the dollar sign in the volume name. Therefore, you might need to declare your network file-name variables large enough to include this extra character. For example:

```
VAR employee : STRING(35) :=
          '\newyork.$payroll.july1990.employee';
    managers : STRING(35) :=
          '\seattle.$disk4.level1.managers    ';
```

Device Names. Your existing program might declare a variable for a Guardian device name. The largest D-series device names are:

Device name without a node name or qualifier	8 characters
Device name without a node name but with a qualifier	17 characters
Network file name without a qualifier	17 characters
Network file name with a qualifier	26 characters

When accessing devices on remote D-series systems in a network, a converted program can use an eight-character network device name (one to seven characters after the dollar sign). A C-series network device name allows a maximum of six characters after the dollar sign. Therefore, you might need to declare your network device names large enough to include this extra character. For example:

```
{ Network device name without a qualifier }

VAR device_name : STRING(17) := '\hamburg.$term001';

{ Network device name with a qualifier }

VAR network_device_name : STRING(26) :=
               '\hamburg.$lineptr.#room025';
```

Process File Names. Your existing program might declare a variable for a C-series process file name. The D-series enhanced interface uses D-series process files names instead of C-series process file names. Use C-series process file names for compatibility with unconverted C-series procedures.

D-series process file names are variable-length string data items with their lengths specified as separate data items. For example:

```
VAR my_process_file   : STRING(47);  { process file name }
```

The PASEXT file contains declarations that you can use for declaring D-series process-file-name variables.

Declaring Process Identifiers

Your existing program might declare a process-ID variable to identify a process (for example, an opener in an opener table). This example uses the process-ID declaration from the TYPES section of the PEXTDECS file:

```
TYPE process_id    = Process_Id_Type;
```

Convert the process-ID variable declaration to a process-handle variable for process-control operations or to a process-descriptor variable for returning information from a Guardian procedure. Use a process-ID variable for compatibility with unconverted C-series procedures.

A process handle is a 10-word (20-character) fixed-length structure. A process descriptor is a specific form of the D-series process file name that always includes the node name and sequence number. For example:

```
VAR my_phandle : ARRAY[1..10] OF INTEGER;  { process handle }

    my_process_desc   : STRING(33);  { process descriptor }
```

The PASEXT file contains declarations that you can use for declaring process-handle and process-descriptor variables.

Avoiding Subvolume Defaulting in Disk File Names

Your existing program might use subvolume defaulting to represent a Guardian disk file name in the form *volume.file-id*. For example:

```
VAR disk_file : STRING(17) := '$diskvol.filename';
```

Avoid subvolume defaulting in your program. If a file name requires the volume name, also include the subvolume name:

```
VAR disk_file : STRING(26) := '$diskvol.subvol.filename  ';
```

Converting Guardian Procedure Calls

Guardian procedures that you might need to convert are procedures that accept or return:

- ☐ A PIN parameter for a high-PIN process
- ☐ A process-ID parameter

D-series procedures use a process handle (which includes the CPU and PIN values) rather than a process ID to identify a process.

If you convert your program to run at a high PIN, then you must replace MYPID procedure calls with calls to the PROCESS_GETINFO_ and PROCESSHANDLE_DECOMPOSE_ procedures. "Converting a Pascal Program to Run at a High-PIN," later in this section, describes how to do this.

Appendix A lists the D-series procedures that supersede C-series procedures. For a description of each procedure, refer to the *Guardian Procedure Calls Reference Manual*.

For examples of Guardian procedure calls, refer to Section 3, "Converting TAL Applications." If you are converting a Pascal program, you must convert the TAL procedure calls shown in Section 3 to Pascal.

Running the Pascal Compiler

When you start the Pascal compiler using the TACL RUN command, TACL calls the PROCESS_CREATE_ procedure to create the Pascal compiler process and the resulting BINSERV and SYMSERV processes at low PINs.

To run the Pascal compiler process (and the BINSERV and SYMSERV processes) at a high PIN, you must use the binder program to set the HIGHPIN object-file attribute to ON in the Pascal compiler object file (provided you have the proper authority to change this file). TACL then runs the Pascal compiler (and the BINSERV and SYMSERV processes) at a high PIN if one is available. For more information about TACL, refer to the *TACL Reference Manual.*

Binding the Run-Time Library

When binding your program on C-series systems, you had the option of selecting either the PASRUN or the PASRUNS run-time support library, depending on the nature of the object that you were binding.

In the D-series version of Pascal, the PASRUN and PASRUNS libraries have been merged into one library called PASLIB. PASLIB resides in the system library by default. If PASLIB does not reside in the system library, you must specify PASLIB when you bind your programs.

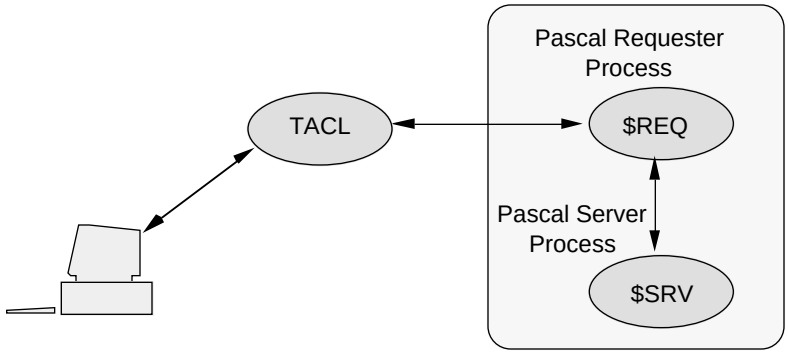
Using the Binder With Converted Object Files

You cannot bind modules that have been compiled with the D-series compiler with modules that have been compiled with a C-series compiler. If you compile any module with the D-series compiler, you must also recompile any other modules that you bind it with.

Converting a Pascal Program to Run at a High PIN

This subsection describes how to convert your Pascal program to run at a high PIN under the D-series operating system. Figure 6-2 shows a typical application. The box shows which processes this part of the conversion applies to. Converting a Pascal program to run at a high PIN applies to both of these processes.

Figure 6-2. Converting a Pascal Program to Run at a High PIN



To convert your program, you must:

- ☐ Set the HIGHPIN object-file attribute (which tells the system that your program can run at a high PIN)
- ☐ Make sure that each library file your program uses also has its HIGHPIN object-file attribute set (and is capable of running at a high PIN)
- ☐ Declare PIN variables large enough to hold high-PIN values
- ☐ Convert MYPID procedure calls into calls to the PROCESS_GETINFO_ and PROCESSHANDLE_DECOMPOSE_ procedures

These topics are described in the following subsections.

Setting the HIGHPIN Object-File Attribute

The HIGHPIN object-file attribute directs the system to run a process at a high PIN if one is available. If a high PIN is not available, the process runs at a low PIN if one is available. You set the HIGHPIN object-file attribute either during compilation using a compiler directive or after compilation using the Binder program.

To set the attribute when you compile your program, specify the HIGHPIN directive in your source code or as a compiler option in the TACL RUN command for the Pascal compiler. The BINSERV program then sets the HIGHPIN attribute in the object file. An example of this directive in your source file is:

```
?HIGHPIN
```

An example of this directive as a compiler option is:

```
10> PASCAL /IN passrc,OUT $s.#paslst,NOWAIT/ pasobj; HIGHPIN
```

You need to specify the HIGHPIN directive only once during a compilation. However, you can specify it any number of times and the compiler will not generate an error.

If you do not set the HIGHPIN attribute when you compile your program, you can set it after compilation using Binder. For a single object file, use the Binder CHANGE command:

```
@CHANGE HIGHPIN ON IN pasobj
```

If you are binding more than one object file into a single target object file, use the Binder SET command to set the HIGHPIN object-file attribute in the target file. For the target object file to have its HIGHPIN object-file attribute set, each constituent object file must also have its HIGHPIN attribute set.

Using a Library File

If your existing program uses a library file, the library file must also have its HIGHPIN object-file attribute set. To determine the current setting of the HIGHPIN attribute for a library file, use the Binder SHOW command:

```
@SHOW SET HIGHPIN FROM libfile
```

If necessary, set this attribute as described in the previous subsection (provided the library file has been converted to support a high-PIN process).

Declaring CPU and PIN Variables

As stated earlier under "Converting Basic Elements of a Pascal Program," your existing program might declare a 16-bit variable for both the CPU and PIN values:

```
VAR cpu_pin     : INTEGER;
```

Or your program might declare an 8-bit variable for a PIN value:

```
VAR pin         : BYTE;
```

Declare all PIN values, including backup-process PIN values, as 16-bit variables. Declare a CPU value as a separate 16-bit variable:

```
VAR cpu, pin    : INTEGER;
```

Converting MYPID Procedure Calls

Your existing program might call the MYPID procedure to obtain its CPU and PIN values:

```
VAR cpu_pin     : INTEGER;

...

cpu_pin := MYPID;
```

If a high-PIN process calls MYPID, a trap condition occurs. You must convert MYPID procedure calls into calls to the PROCESSHANDLE_DECOMPOSE_ procedure.

The PROCESSHANDLE_DECOMPOSE_ procedure requires a process handle as an input parameter. If you do not know the process handle of your process, first call the PROCESSHANDLE_GETMINE_ procedure. Then pass the result to PROCESSHANDLE_DECOMPOSE_ , which returns the CPU and PIN values as separate integer values. For example:

```
VAR my_phandle     : ARRAY[1..10] OF INTEGER;
    my_cpu, my_pin : INTEGER;
    status         : INTEGER;

{ Return my process handle. }

status := PROCESSHANDLE_GETMINE_(my_phandle);

IF status <> 0 THEN error_routine (status);

{ Return my CPU and PIN values. }

status := PROCESSHANDLE_DECOMPOSE_(my_phandle,
                                   my_cpu,
                                   my_pin);
IF status <> 0 THEN error_routine (status);
```

Your existing program might also call the MYPID procedure within another Guardian procedure call (for example, in a SETMODE function 11, GETCRTPID, or PROCESSINFO call). This example shows MYPID in a SETMODE (function 11) procedure call:

```
status := SETMODE (file_number,
                   11,
                   MYPID,   { Set to the CPU and PIN. }
                   0,
                   previous_owner);
```

For SETMODE function 11, you are not required to set the *parameter_1* value to the CPU and PIN values. Instead, set *parameter_1* to any positive value:

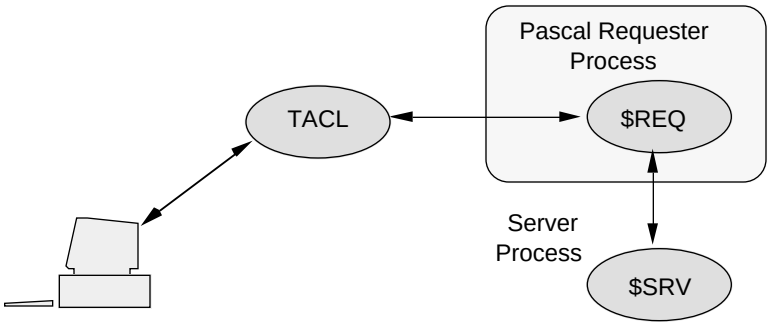
```
status := SETMODE (file_number,
                   11,
                   1,  { Set to any positive value. }
                   0,
                   previous_owner);
```

Creating a High-PIN Process

You can create a high-PIN process programmatically or interactively. This subsection describes how to programmatically create a high-PIN process. For information on how to interactively create a high-PIN process using TACL, see Section 7, "Converting TACL Programs."

Figure 6-3 shows the processes involved in converting a typical application. The steps described in this subsection apply to the requester process $REQ, which creates the high-PIN server process $SRV.

Figure 6-3. Converting a Pascal Program to Create a High-PIN Process



Your existing program might create a new process using the NEWPROCESS[NOWAIT] procedure:

```
IMPORT BEGIN
?SOURCE $SYSTEM.SYSTEM.PEXTDECS (TYPES,
?                                NEWPROCESS,
?                                Guardian_OPEN,
?                                Guardian_READ,
?                                Guardian_WRITE,
?                                Guardian_CLOSE)
END; { IMPORT }
...

int program_file[12];
int home_terminal[12];
int process_id[4];
int process_name[4];
int err_return, priority, memory_pages, cpu;
int error_info[2];
int status;

...
```

```
status := NEWPROCESS (program_file,
                      priority,
                      memory_pages,
                      cpu,
                      process_id,
                      err_return,
                      process_name,
                      home_terminal,
                      { flags  } 0,
                      { job_id }  ,
                      error_info);
```

On a D-series system, the NEWPROCESS[NOWAIT] procedure can create only a low-PIN process. The D-series operating system provides the PROCESS_CREATE_ procedure to create a new low-PIN or high-PIN process in a waited or nowait manner.

The following procedure creates a new process in a waited manner. The system returns the results in the returned value *error* and *error_detail* parameter:

```
IMPORT BEGIN
?SOURCE $SYSTEM.SYSTEM.PEXTDECS (TYPES,
?                                PROCESS_CREATE_,
?                                FILE_OPEN_,
?                                Guardian_READ,
?                                Guardian_WRITE,
?                                FILE_CLOSE_)
END; { IMPORT }
...

char program_file[36];
char process_handle[20];

int pf_length;
int err, priority, memory_pages, cpu, error_detail;
...
error := PROCESS_CREATE_(program_file:pf_length,
                         { library_file:lf_length  } ,
                         { swapfile:sf_length      } ,
                         { ext_swapfile:esf_length } ,
                         priority,
                         cpu,
                         process_handle,
                         error_detail);
```

If your program creates a new process in a nowait manner, the system returns the results in system message -102 (PROCESS_CREATE_ completion), which is analogous to system message -12 (NEWPROCESSNOWAIT completion). You read system message -102 from $RECEIVE using the READ or READUPDATE procedure.

For additional information on creating processes in a nowait manner, creating processes at a low PIN, and other information about the PROCESS_CREATE_ procedure, refer to Section 3, “Converting TAL Applications.”

Opening and Communicating With a High-PIN Server

Your existing program might be a requester that communicates with a server. For example, you might open a server, send it a request, and then process its reply. You might also open a server for a backup requester if your program is running as a process pair.

The degree of conversion you need to perform depends on whether your server is named or unnamed, and, if the server is named, on how long the name is. Your options are as follows:

- If the server is local and named or if the server is remote with a name of five characters or less (including the dollar sign), then no conversion is necessary. You can still open the high-PIN server using the Guardian C-series-compatible OPEN procedure. See Appendix C, "System Compatibility," for further information on communicating with a named high-PIN process.
- If the server is remote and has a six-character name, then you need to first convert your requester to run at a high PIN as described under "Converting a Pascal Program to Run at a High PIN" earlier in this section, and then complete the conversion as described under "Communicating with a High-PIN Server" and "Monitoring a High-PIN Server," later in this section. See Appendix C, "System Compatibility" for further information on communicating with a named high-PIN process.
- If the server is unnamed, then you have the following options:
  - Set the RUNNAMED object-file attribute in the server so that the system provides a name for the server, and pass the system-assigned name to the requester; for example, in a DEFINE or an ASSIGN. See "Setting the RUNNAMED Object-File Attribute" later in this section for details.
  - Convert the requester to run at a high PIN as described under "Converting a Pascal Program to Run at a High PIN" earlier in this section, and then complete the conversion as described under "Communicating with a High-PIN Server" and "Monitoring a High-PIN Server," later in this section.

For information about converting a server to monitor a high-PIN requester process, including maintaining an opener table, refer to "Being Opened by and Communicating With a High-PIN Requester," later in this section.

Setting the RUNNAMED Object-File Attribute

The RUNNAMED object-file attribute causes a process to run as a named process even if you do not provide a name for it. Thus, a process can run at a high PIN under the D-series operating system and still be opened by an unconverted process using the Guardian OPEN procedure. For more information about how an unconverted process running on a D-series system can communicate with a named high-PIN process, refer to Appendix C, "System Compatibility."

You set the RUNNAMED object-file attribute either during compilation using a compiler directive or after compilation using the Binder program.

To set the attribute when you compile your program, specify the RUNNAMED directive in your source code or as a compiler option in the TACL RUN command for the Pascal compiler. The BINSERV program then sets the RUNNAMED attribute in the object file. An example of this directive (with the HIGHPIN directive) in your source file is:

```
?HIGHPIN, RUNNAMED
```

An example of this directive as a compiler option is:

```
10> PASCAL /IN passrc, ... / pasobj; HIGHPIN, RUNNAMED
```

You need to specify the RUNNAMED directive only once during a compilation. However, you can specify it any number of times and the compiler will not generate an error.

If you do not set the RUNNAMED attribute when you compile your program, you can set it after compilation using Binder. For a single object file, use the Binder CHANGE command:

```
@CHANGE RUNNAMED ON IN pasobj
```

If you are binding more than one object file into a single target object file, use the Binder SET command to set the RUNNAMED object-file attribute. If any of the constituent object files used to build the target file has the RUNNAMED object-file attribute set, Binder sets this attribute in the target object file.

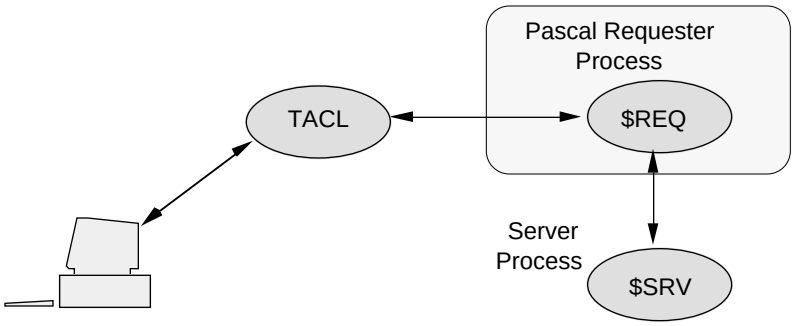
Communicating With a High-PIN Server

A requester can open and communicate with a high-PIN named server by opening the server using the Guardian OPEN procedure. However, you must convert your requester to open the server using the FILE_OPEN_ procedure if the server:

- ☐ Is unnamed
- ☐ Is on a remote D-series system and has a six-character name (a dollar sign and five alphanumeric characters)

Figure 6-4 shows the processes involved in converting this part of a typical application. The steps in this subsection apply to the requester process $REQ.

Figure 6-4. Converting a Pascal Requester to Communicate With a High-PIN Server



This subsection discusses converting the following operations:

- ☐ Opening and closing the high-PIN server
- ☐ Opening and closing the high-PIN server for a backup process
- ☐ Sending requests to the high-PIN server

Opening a High-PIN Server

Your requester might open the server using the Guardian OPEN procedure:

```
VAR server_name: ARRAY[1..12] = OF INTEGER

...

server_name := "$SRV                      ";
status := Guardian_OPEN (server_name,
                         server_file_number,
                         nowait_depth,
                         sync_depth);
```

Convert your requester to open the high-PIN server using the FILE_OPEN_ procedure. The FILE_OPEN_ procedure requires a variable-length string for the server file-name input parameter rather than the 12-word internal-format file name.

Note If the *file-name* input parameter is incomplete (that is, not fully qualified), FILE_OPEN_ uses the current settings, including the system name, in the =_DEFAULTS DEFINE for the unspecified parts.

FILE_OPEN_ also accepts a DEFINE name that represents a valid file name in this format.

FILE_OPEN_ accepts an integer *options* parameter to specify certain file characteristics. The *options* bit positions represent these options:

options Bit Position	Description
0	Allow unstructured access for a disk file (must be 0 for other files and devices)
1	Execute a nowait open
2	Do not execute an update when the file is opened
3	Use any available file number for backup open (0 means use the same file number as in the primary open)
4 through 13	Reserved; must be 0
14	Receive C-series system messages ($RECEIVE only)
15	Do not receive process open and close system messages ($RECEIVE only)

The ZSYSPAS file contains constant declarations that you can use with the *options* parameter.

If you started the server using the PROCESS_CREATE_ procedure, you can use the PROCESS_CREATE_ process-descriptor output parameter directly in the FILE_OPEN_ procedure call (shown below as the *server_name* parameter). Refer to "Creating and Managing a High-PIN Process" earlier in this section for details.

```
error := FILE_OPEN_(server_name:server_length,
                    server_file_number,
                    exclusion_mode,
                    nowait_operations,
                    sync_depth,
                    options);
```

If you open the server using the nowait open option, you must call the AWAITIO[X] procedure to complete the open. To determine the *error* and *options* values, call the FILE_GETINFOLIST_ procedure and check the items specified by ZSYS_VAL_FINF_LASTERROR and ZSYS_VAL_FINF_OPENOPTS, respectively (provided you use the ZSYSPAS file).

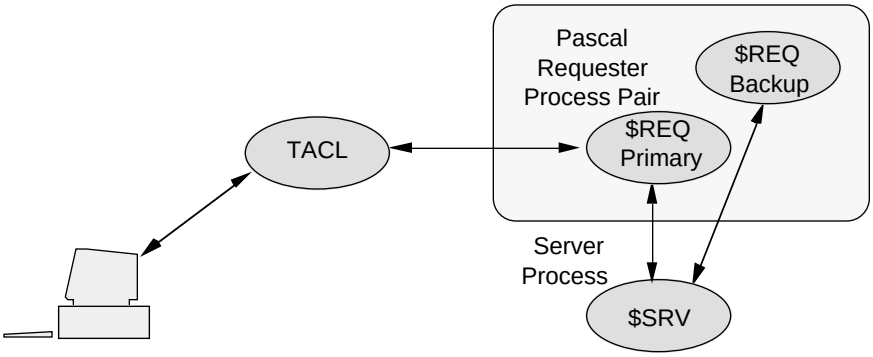
Opening a High-PIN Server for a Backup Requester Process

If your requester is running as a process pair, it might open the server for its backup process using the CHECKOPEN procedure:

```
status := CHECKOPEN (server_name,
                     server_file_number,
                     nowait_depth,
                     sync_depth,
                     { seq_block_buffer } ,
                     { buffer_length    } ,
                     back_error);
```

Figure 6-5 shows a requester process pair and a server process.

Figure 6-5. Opening a High-PIN Server for a Backup Process



Convert your requester to open the high-PIN server for its backup process using the FILE_OPEN_CHKPT_ procedure. To identify the high-PIN server, FILE_OPEN_CHKPT_ requires the file number returned by the FILE_OPEN_ procedure call in the primary process. The system returns a file-system error (if a file-system error occurs) in the returned value *error* and the status of the backup open in an output parameter, which is the *backup_open_status* parameter in this example:

```
error := FILE_OPEN_CHKPT_(server_file_number,
                          backup_open_status);
```

If you opened the server using the nowait open option, you must call the AWAITIO[X] procedure to complete the open. To determine the *error* and *backup_open_status* values, call the FILE_GETINFOLIST_ procedure and check the items specified by ZSYS_VAL_FINF_LASTERROR and ZSYS_VAL_FINF_LASTERRORDETAIL, respectively (provided you use the ZSYSPAS file).

Sending a Request to a High-PIN Server

Your requester might send a request to a high-PIN server using the Guardian WRITE[X] or WRITEREAD[X] procedure:

```
status := WRITEREADX (server_file_number,
                      sbuffer,
                      write_count,
                      read_count,
                      count_read);
```

Your Guardian WRITE[X] or WRITEREAD[X] procedure call should not require any changes to send a request to a high-PIN server.

Closing a High-PIN Server

Your requester might close the server using the Guardian CLOSE procedure:

```
status := Guardian_CLOSE (server_file_number);
```

You can close a high-PIN server using either the Guardian CLOSE or FILE_CLOSE_ procedure:

```
error := FILE_CLOSE_(server_file_number);
```

Closing a High-PIN Server for a Backup Requester Process

Your requester might close the server for the backup process using the CHECKCLOSE procedure:

```
status := CHECKCLOSE(server_file_number);
```

You can close the server for the backup process using either the Guardian CLOSE procedure or the FILE_CLOSE_CHKPT_ procedure:

```
error := FILE_CLOSE_CHKPT_(server_file_number);
```

Monitoring a High-PIN Server

If your program monitors a high-PIN server, you must convert the following operations:

- ☐ Opening and closing $RECEIVE
- ☐ Reading process-deletion and status-change messages
- ☐ Using the CHILD_LOST_ procedure

The following paragraphs describe how to convert these operations. These steps also can apply to any creator process that monitors a process that it has created.

Opening $RECEIVE

Your requester might open $RECEIVE using the Guardian OPEN procedure:

```
VAR receive_name: ARRAY [1..12] OF INT;
CONST read_open_close_msgs = 8#40000;

...

receive_name := "$RECEIVE               ";
status := Guardian_OPEN (receive_name,
                         receive_file_number,
                         read_open_close_msgs,
                         receive_depth);
```

Convert your requester to open $RECEIVE using the FILE_OPEN_ procedure. Use a file-name string for the $RECEIVE file name instead of the internal file-name format. Specify the length as a separate integer value.

Bit 14 of the *options* parameter must be zero (which is the default value) for the system to send D-series system messages to $RECEIVE; otherwise, the system sends C-series system messages to $RECEIVE for the requester.

An example of a FILE_OPEN_ procedure call for $RECEIVE is:

```
CONST receive_name_length = 8;

VAR receive_name: ARRAY [1..receive_name_len] OF CHAR

...

{ Open $RECEIVE to read D-series system messages }

receive_name := "$RECEIVE";
  error := FILE_OPEN_(receive_name:receive_name_length,
                      receive_file_number,
                      { access_mode       } ,
                      { exclusion_mode    } ,
                      { nowait_operations } ,
                      receive_depth,
                      options);
```

Reading System Messages From $RECEIVE

Your requester might read system messages from $RECEIVE using the Guardian READ[X] or READUPDATE[X] procedure:

```
{ Message buffer (200 bytes) }
VAR message_buffer : ARRAY[1..200] OF CHAR;

...

read_count := 200;

status := READX (receive_file_number,
                 message_buffer,
                 read_count,
                 bytes_read);
```

The lengths shown for each system message are subject to change. In a future release, Tandem might add new fields to the end of a system message (while maintaining the layout of the existing fields). Therefore, use a Guardian READ[X] or READUPDATE[X] message buffer at least 250 bytes in length. Also, use a *read_count* parameter of 250 bytes.

If you use the ZSYSPAS file, use the ZSYS_VAL_SMSG_LEN constant declaration to specify the system message length in bytes. If you work in words you can use the ZSYS_VAL_SMSG_WLEN constant declaration instead.

```
VAR message_buffer : ARRAY [1..ZSYS_VAL_SMSG_LEN] OF CHAR;

...

read_count := ZSYS_VAL_SMSG_LEN;

status := READX (receive_file_number,
                 message_buffer,
                 read_count,
                 bytes_read);
```

The ZSYSPAS file also contains structures that you can use when your requester reads system messages.

Reading Process-Deletion System Messages. Your requester might monitor a server process by reading these process-deletion system messages from $RECEIVE:

-2 CPU down: named process deletion
-5 Process normal deletion: stop
-6 Process abnormal deletion: abend

Convert your requester to read and process the D-series system message -101 (Process deletion), which supersedes all the above messages.

Reading Status-Change System Messages. Your requester might monitor a server process by reading these status-change system messages from $RECEIVE:

-2 CPU down: local CPU failure after process called MONITORCPUS
-8 Change in status of network node

Continue to read system message -2. Then, convert your requester to read these new status-change messages, all of which supersede system message -8:

-100 Remote CPU down
-110 Loss of communication with node
-113 Remote CPU up

To receive system messages -100, -110, and -113, first call the MONITORNET procedure with the *enable* parameter set to 1.

Processing System Messages Using the CHILD_LOST_ Procedure

Your requester might call a user-written routine to determine whether a process-deletion or status-change message affects the server.

You might convert your requester to call the new CHILD_LOST_ procedure. The CHILD_LOST_ procedure accepts the process handle of a process you are monitoring and either a C-series (-2, -5, -6, or -8) or D-series (-2, -100, -101, -110, or -113) process-deletion or status-change system message:

```
error := CHILD_LOST_(message:message_length,
                     process_handle);
```

The CHILD_LOST_ *error* returned value indicates whether the process (or process pair) is lost:

0 The process (or process pair) is not lost.
4 The process (or process pair) is lost.

Note System message -101 (Process deletion) contains the process handle and process descriptor of the process that terminated. If a named process (or process pair) has terminated, this is the last opportunity for you to save the process name of the process (or process pair).

Closing $RECEIVE

Your requester might close $RECEIVE using the Guardian CLOSE procedure:

```
status := Guardian_CLOSE (receive_file_number);
```

You can close $RECEIVE using either the Guardian CLOSE or FILE_CLOSE_ procedure:

```
error := FILE_CLOSE_(receive_file_number);
```

Being Opened by and Communicating With a High-PIN Requester

This subsection describes how to convert a Pascal server to communicate with a high-PIN requester. Whether you need to convert the server process depends in part on whether the server tracks its openers. If the server does keep track of its openers, you should enable the server to run at a high PIN as described in "Converting a Pascal Program to Run at a High PIN," earlier in this section, and then convert the server as described under "Converting a Server," later in this subsection.

If the server does not track its openers, or if you choose not to perform the conversion, then you can keep the server process at a low PIN and not convert it, except for setting the HIGHREQUESTERS object-file attribute as described under "Setting the HIGHREQUESTERS Attribute to Allow High-PIN Openers," later in this subsection. Setting this attribute enables a high-PIN requester to open a low-PIN server.

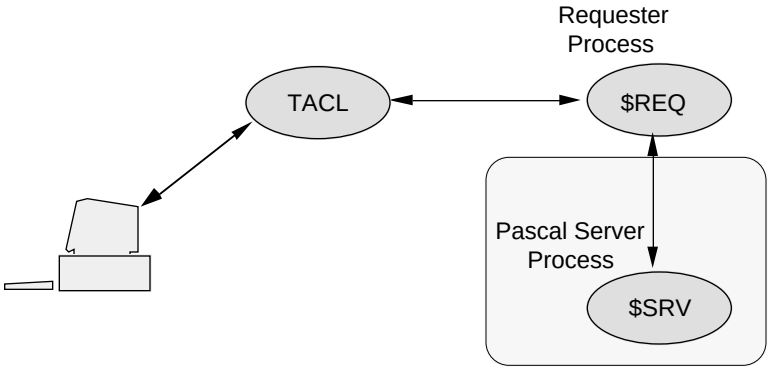
Converting a Server

If your server process tracks its openers, you must convert the following parts of your program:

- ☐ Defining an opener table
- ☐ Opening $RECEIVE
- ☐ Reading D-series system messages from $RECEIVE
- ☐ Getting information about system messages
- ☐ Processing system messages
- ☐ Replying to a system message
- ☐ Using the OPENER_LOST_ procedure to maintain an opener table

Figure 6-6 shows the processes involved in converting an application. The steps described in this subsection apply to the server process $SRV.

Figure 6-6. Converting a Pascal Server to Communicate With a High-PIN Requester



Defining an Opener Table

If your server tracks its openers, it might define an opener table that uses a process ID to identify an opener (primary process opener and backup process opener):

```
TYPE opener = RECORD
               primary_process_id:  ARRAY [1..4] OF INTEGER;
               primary_file_number: INTEGER;
               backup_process_id:   ARRAY [1..4] OF INTEGER;
               backup-file_number:  INTEGER;
              END;

TYPE opener_table = RECORD
               current_count: INTEGER;
               opener_list: ARRAY [1..max_openers] OF opener;
              END;
```

Convert your opener table to identify an opener using a process handle rather than a process ID. To use the OPENER_LOST_ procedure (which is described later in this subsection) to manage your opener table, define the table as follows:

- ☐ Use a process handle to identify both a primary-process and backup-process opener.
- ☐ Declare the process-handle field for the backup-process opener immediately after the process-handle field for the primary-process opener (that is, the fields must be stored in a 20-word contiguous part of an entry).
- ☐ Declare table entries as fixed length and contiguous.
- ☐ Do not store variable-length items in the table. If necessary, save a pointer in the table to a variable-length item.
- ☐ Set the process handles for primary and backup openers in unused entries to null values (all -1s).

An example of an opener table that the OPENER_LOST_ procedure can process is:

```
TYPE opener = RECORD
               primary_process_handle:
                 ARRAY [1..ZSYS_VAL_PHANDLE_WLEN] OF INTEGER;
               backup_process_handle:
                 ARRAY [1..ZSYS_VAL_PHANDLE_WLEN] OF INTEGER;
               primary_file_number: INTEGER;
               backup_file_number:  INTEGER;
              END;

TYPE opener_table = RECORD
               current_count: INTEGER;
               opener_list: ARRAY [1..max_openers] OF opener;
              END;
```

Opening $RECEIVE

Your server might open $RECEIVE using the OPEN procedure with bit 1 of the *flags* parameter set to 1 (*flags* = 8#40000). This allows you to receive system messages such as -30 (Process open) and -31 (Process close):

```
VAR   receive_name: ARRAY[1..12] OF INTEGER;
CONST read_open_close_msgs = 8#40000;

...
receive_name := "$RECEIVE               ";
status := Guardian_OPEN (receive_name,
                         receive_file_number,
                         read_open_close_msgs, { octal 40000}
                         receive_depth);
```

Convert your server to open $RECEIVE using the FILE_OPEN_ procedure:

1. Use a file-name string for the $RECEIVE file name instead of the internal file-name format. Specify the length as a separate integer value.
2. Make sure that bit 15 of the FILE_OPEN_ *options* parameter is zero (the default value). If this bit is not zero, system messages such as -103 (Process open) and -104 (Process close) are not sent to $RECEIVE.
3. Make sure that bit 14 of the FILE_OPEN_ *options* parameter is zero (the default value) so that the system sends D-series system messages to $RECEIVE. If this bit is not zero, the system sends C-series system messages to $RECEIVE.
4. Set any other FILE_OPEN_ input parameters as required and call the procedure:

```
CONST receive_name_length = 8;
VAR  receive_name: ARRAY [1..receive_name_length] OF CHAR;

...
{ Open $RECEIVE to read D-series system messages }

receive_name := "$RECEIVE";
error := FILE_OPEN_(receive_name:receive_name_length,
                    receive_file_number,
                    { access_mode        } ,
                    { exclusion_mode     } ,
                    nowait_operations,
                    receive_depth);
```

If you open $RECEIVE using the FILE_OPEN_ procedure, the system assumes that you support high-PIN requesters (provided bit 14 of the *options* parameter is zero). You do not need to explicitly set the HIGHREQUESTERS object-file attribute in your server's object file.

When you close $RECEIVE, use either the Guardian CLOSE or FILE_CLOSE_ procedure.

Reading System Messages From $RECEIVE

Your server might read system messages from $RECEIVE using the Guardian READ[X] or READUPDATE[X] procedure:

```
{ Message buffer (200 bytes) }
VAR message_buffer: ARRAY [1..200] OF CHAR;

...
read_count := 200;

status := READX (receive_file_number,
                 message_buffer,
                 read_count,
                 bytes_read);
```

The lengths shown for each system message are subject to change. Use a Guardian READ[X] or READUPDATE[X] message buffer at least 250 bytes in length. Also, use a *read_count* parameter value of 250 bytes.

If you use the declarations in the ZSYSPAS file, use the ZSYS_VAL_SMSG_LEN constant for the system-message length in bytes or the ZSYS_VAL_SMSG_WLEN constant for the length in words:

```
VAR message_buffer: ARRAY [1..ZSYS_VAL_SMSG_LEN] OF CHAR;

...

read_count := ZSYS_VAL_SMSG_LEN;

status := READX (receive_file_number,
                 message_buffer,
                 read_count,
                 bytes_read);
```

Getting Information About System Messages

Your server might call the RECEIVEINFO or LASTRECEIVE procedure to obtain information about the last message read from $RECEIVE:

```
status := RECEIVEINFO (process_id,
                       message_tag,
                       sync_id,
                       file_number,
                       read_count,
                       io_type);
```

Convert the RECEIVEINFO or LASTRECEIVE call into a call to the FILE_GETRECEIVEINFO_ procedure:

```
{ Return information about the last message }

error := FILE_GETRECEIVEINFO_(message_info);
```

FILE_GETRECEIVEINFO_ returns information in the 17-word *message_info* parameter, which has the format shown in Table 6-1. The ZSYSPAS file contains a structure that you can use for the *message_info* format.

Table 6-1. FILE_GETRECEIVEINFO_ message_info Parameter Format

Word	Description
0	I/O type for the message: 0 = A system message was sent. 1 = The sender called Guardian WRITE[X] procedure. 2 = The sender called Guardian READ[X] procedure. 3 = The sender called WRITEREAD[X] procedure.
1	The maximum reply count in bytes
2	The message tag identifying the message
3	The file number for the message
4 through 5	The sync ID for the message
6 through 15	The process handle of the process sending the message
16	The *open_label* from a previous reply (or -1 if unavailable or for a C-series message)

Reading and Processing Open and Close System Messages

To monitor an opener, your server might read the C-series -30 (Process open) and -31 (Process close) system messages from $RECEIVE.

To monitor a high-PIN process, convert your server to read the D-series -103 (Process open) and -104 (Process close) system messages. When your server is opened or closed by a process pair, it receives a process-open or process-close message from each process of the pair.

If you call the RECEIVEINFO or LASTRECEIVE procedure to obtain information about the process-open or process-close message, convert the call into a call to the FILE_GETRECEIVEINFO_ procedure as described under “Getting Information About System Messages,” earlier in this section.

After calling FILE_GETRECEIVEINFO_, update your opener table using the process handle rather than the process ID to identify the opener.

System Message -103 (Process Open). Check bit 15 of *sysmsg*[7] of the process-open message (or ZSYS_DDL_SMSG_OPEN.Z_FLAGS if you use the ZSYSPAS file), which indicates whether the opener is a primary or backup process:

- ☐ Primary open (*sysmsg*[7] bit 15 is 0): Add an entry in your opener table for the process.
- ☐ Backup open (*sysmsg*[7] bit 15 is 1): Process a backup open as follows:
  1. Get the process handle for the primary opener from the process-open system message (-103). This process handle is in *sysmsg*[8] for ten words (or the ZSYS_DDL_SMSG_OPEN.Z_PRIMARY field if you use the ZSYSPAS file).
  2. Use the process handle to search your opener table for the corresponding primary-process open entry. If you find this entry but there is no backup open yet (the backup process handle is null), add the backup process handle to the table entry.
  3. If the primary-process open entry is not found, reject the backup open with a file-system error greater than 9.

System Message -104 (Process Close). Delete the opener-table entry for this process. You should receive a process-close message from each process of a process pair.

Reading and Processing Status-Change Messages

If one of your openers has a CPU failure, or if its system fails or becomes partitioned from your system because of a network failure, you do not receive a process-close message (-31). Therefore, when maintaining an opener table, your server might read and process these status-change messages:

-2	CPU down: local CPU failure after the process called MONITORCPUS
-8	Change in status of network node

Continue to read system message -2. In addition, read these new status-change messages (all of which supersede C-series system message -8):

-100	Remote CPU down
-110	Loss of communication with node
-113	Remote CPU up

To receive system messages -100, -110, and -113, first call the MONITORNET procedure with the *enable* parameter set to 1.

Replying to a System Message

Your server might reply to a system message using the Guardian REPLY[X] procedure:

```
status := REPLYX (reply_buffer,
                  write_count,
                  count_written,
                  message_tag,
                  error_return);
```

Replying to System Message -103 (Process Open). The D-series system supports returning a label value in the reply to a system message -103 (Process open). Typically, an opener-table index gets sent in this way. This label then appears in the *open_label* field of future FILE_GETRECEIVEINFO_ procedure calls that provide information about messages received from the same requester. To support this feature, the file system expects a reply buffer with a length of 0 to 4 bytes; otherwise, the open in the requester returns an error.

Your server might reply to an Open message as follows:

```
write_count := any_valid_integer;
CALL REPLYX(reply_buffer,
            write_count,
            ! count_written ! ,
            ! message_tag   ! ,
            error_return);
```

To make use of the *open_label* field in the FILE_GETRECEIVEINFO_ procedure, you must convert your code to reply to the Open message as follows:

```
reply_buffer[0] := -103;
reply_buffer[1] := open_label_value;
write_count := 4;
CALL REPLYX (reply_buffer,
             write_count,
             { count_written } ,
             { message_tag   } ,
             error_return);
```

If you do not want to use the *open_label* field, you still need to be sure that the reply buffer has a length of 0 to 4 bytes. Convert your server as follows:

```
write_count := 0;
CALL REPLYX (reply_buffer,
             write_count,
             { count_written } ,
             { message_tag   } ,
             error_return);
```

Replying to an Unknown System Message. Your server should be able to handle an unknown system message. If the first word of a message contains an unknown message number, call the REPLY[X] procedure with an error indication of 2 (invalid operation):

```
status := REPLYX ({ reply_buffer  } ,
                  { write_count   } ,
                  { count_written } ,
                  { message_tag   } ,
                  invalid_operation);   /* Value is 2 */
```

Using the OPENER_LOST_ Procedure to Maintain an Opener Table

After receiving a status-change message, your server might call one or more routines to maintain its opener table.

You might want to use the OPENER_LOST_ procedure to maintain your opener table. OPENER_LOST_ determines whether a status-change message affects your opener table and updates the appropriate table entry if an opener was lost.

OPENER_LOST_ accepts a C-series (-2 or -8) or D-series (-2, -100, -110, or -113) status-change message and searches your opener table for any processes affected by the message. If OPENER_LOST_ determines that an opener has been lost, it updates the opener-table entry and returns the index of the entry and an *error* value. The *error* value indicates the reason for the opener-table change:

error Value	Reason
4	A backup process opener is lost
5	A primary process opener is lost; the backup process is now the primary process
6	The primary process and backup process (if it exists) openers for a table entry are lost; the table entry is now free

When OPENER_LOST_ returns an *error* value of zero, processing is complete for the message.

To process all entries in your opener table for a status-change message, set up a loop similar to the one shown below. The opener table must be defined as described under "Defining an Opener Table," earlier in this subsection.

```
done  :=  0;    { Set control for start of loop }
index := -1;    { Set index for start of loop }

REPEAT
   error := OPENER_LOST_(message:message_length,
                         opener_table.opener_list,
                         index,
                         opener_table.current_count,
                         $LEN(opener_table.opener_list));

  CASE error OF
     4:  { Processing for lost backup opener }
     5:  { Processing for lost primary opener }
     6:  { Processing for lost opener }
         { (primary and backup for a process pair) }
     OTHERWISE: done = -1   { Processing is finished or }
                            { error occurred }
  END
UNTIL done = 0
```

Setting the HIGHREQUESTERS Attribute to Allow High-PIN Openers

The HIGHREQUESTERS object-file attribute allows a process to support requests from high-PIN requesters. Use this attribute only for a Pascal main program. You can set the HIGHREQUESTERS object-file attribute by including a compiler directive in your source file, or you can set it after you have finished converting your source code either using a compiler option or after compilation using the Binder program.

To set the attribute when you compile your program, specify the HIGHREQUESTERS directive in your source code or as a compiler option in the TACL RUN command for the Pascal compiler. The BINSERV program then sets the HIGHREQUESTERS attribute in the object file. An example of this directive in your source file is:

```
?HIGHREQUESTERS
```

An example of this directive as a compiler option is:

```
10> PASCAL / IN passrc, ... / pasobj; HIGHREQUESTERS
```

You need to specify the HIGHREQUESTERS directive only once during a compilation. If your program file copies source code from another file, specify the HIGHREQUESTERS directive only in the program file that contains the main program; do not specify the directive in the other file (or files).

If you do not set the HIGHREQUESTERS attribute when you compile your program, you can set it after compilation using Binder. For a single object file, use the Binder CHANGE command:

```
@CHANGE HIGHREQUESTERS ON IN pasobj
```

If you are binding more than one object file into a single target object file, use the Binder SET command to set the HIGHREQUESTERS object-file attribute. For Binder to set the HIGHREQUESTERS object-file attribute in a target object file, the object file containing the main program must have this object-file attribute set.

For more information about the HIGHREQUESTERS object-file attribute, refer to "Allowing Opens by High-PIN Requesters" in Appendix C, "System Compatibility."

7 Converting TACL Programs

Whether you use TACL commands and functions interactively or write TACL programs, you might need to make some changes to take advantage of the D-series enhanced interface.

This section describes how to convert your TACL programs. The topics covered are:

- ☐ How to declare and use TACL variables such as CPU and PIN variables
- ☐ How to create and manage high-PIN processes, including how to obtain completion information
- ☐ How to handle D-series variances in the information returned by some TACL functions
- ☐ How to obtain information about file and record locks.

Section 8, "Converting Other Parts of an Application," contains information about converting other parts of a TACL program. For further information about TACL programming, including new features, see the *TACL Reference Manual* and the *TACL Programmer's Guide.*

Declaring and Using TACL Variables

This subsection describes how to change TACL declarations for:

- ☐ File-system error numbers
- ☐ CPU and PIN variables
- ☐ Process identifiers

This subsection also provides information on how to:

- ☐ Avoid subvolume defaulting
- ☐ Convert between process handles and process file names

Declaring File-System Error Numbers

File-system error numbers under the C-series operating system are all in the range 0 through 255. The C-series-compatible interface retains the same set of error numbers. The D-series enhanced interface, however, extends this range beyond 255.

Your existing program might use a BYTE field in a STRUCT for storing a returned file-system error number. For example:

```
[#DEF error^info STRUCT
  BEGIN
     BYTE file^error;
     ...
  END;
]
```

Change the BYTE field to an INT field to allow for the extended range of file system error numbers. For example:

```
[#DEF error^info STRUCT
  BEGIN
     INT file^error;
     ...
  END;
]
```

Declaring CPU and PIN Variables

The C-series-compatible interface allows for PIN values up to 255. Using the D-series enhanced interface, you can still use low PINs in the range 1 through 254, or you can use high PINs in the range 256 up to the limit for the CPU.

Your existing program might declare BYTE or INT data types in STRUCTs for CPU and PIN values. The following example uses BYTE data types:

```
[#DEF cpu^and^pin STRUCT
  BEGIN
     BYTE cpu;
     BYTE pin;
  END;
]
```

To allow for high-PIN processes, convert your program to use INT data types for both CPU and PIN values. For example:

```
[#DEF cpu^and^pin STRUCT
  BEGIN
     INT cpu;
     INT pin;
  END;
]
```

Declaring Process Identifiers

The C-series interface uses 4-word process IDs to identify processes. The D-series enhanced interface instead uses 20-byte process handles or process descriptor strings, depending on what the identifier is used for.

Your existing program might declare a STRUCT containing a process ID. For example:

```
[#DEF process^identifier STRUCT
  BEGIN
     INT   process^id(0:3);
     BYTE  cpu;
     BYTE  pin;
  END;
]
```

You need to change the declaration to contain either a process handle (for process-control operations) or a process descriptor (used by Guardian procedures to return the identity of a process).

A process handle is a series of 10 numbers in the range 0 through 65535 separated by periods. You declare a process handle using the new PHANDLE data type.

A process descriptor is represented by a string of at most 29 bytes for an unnamed process or 33 bytes for a named process. For example:

```
[#DEF process^information STRUCT
  BEGIN
     PHANDLE  process^handle;
     CHAR     process^descriptor(0:32);
     INT      cpu;
     INT      pin;
  END;
]
```

If your existing program is part of a DSM application and uses SPI buffers, the process IDs might be identified by a ZSPI-TYP-CRTPID token type.

Convert your application to use either a process-descriptor token or a process-handle token instead of the process identifier. You need to use a process descriptor when the token contains information returned from a Guardian procedure. You can use the token type ZSPI-TYP-STRING for a process descriptor.

For process-control information, the token contains a process handle. You should use the ZSPI-TDT-PHANDLE token data type.

For details of converting DSM applications, see Section 8, "Converting Other Parts of an Application."

Avoiding Subvolume Defaulting

Your existing program might use subvolume defaulting to represent a Guardian disk file name in the form *volume.file-id*. For example:

```
$MYVOL.MYFILE
```

If you are using the D-series programmatic interface, you must explicitly specify the subvolume. If a file name requires the volume name, also include the subvolume name:

```
$MYVOL.MYSUBVOL.MYFILE
```

Your existing TACL program might use the #FILEINFO built-in function to obtain the volume part of the current default values as follows:

```
#FILEINFO /VOLUME/ [#DEFAULTS]
```

#FILEINFO now requires a file identifier which is not supplied by the defaults. You must therefore supply a file ID. Replace the call with:

```
#FILEINFO /VOLUME/ [#DEFAULTS].X
```

Converting Between Process Handles and Process File Names

A new built-in function, #CONVERTPHANDLE, converts process handles to process descriptors and process file names to process handles.

Converting Process Handles to Process Descriptors

You can convert a process handle into a process descriptor using #CONVERTPHANDLE as shown in the following example:

```
[#DEF process^name STRUCT
  BEGIN
     CHAR process^file^name(0:46);
  END;
]

[#DEF process^handle STRUCT
  BEGIN
     PHANDLE proc^handle;
  END;
]

...
#SET process^name &
     [#CONVERTPHANDLE /PROCESSID/ process^handle]
```

Converting Process File Names to Process Handles

The following example converts a process file name into a process handle using the same STRUCT definitions as the previous example:

```
#SET process^handle [#CONVERTPHANDLE /INTEGERS/ process^name]
```

Creating and Managing a High-PIN Process

This subsection describes:

- How to use TACL to run processes at a high PIN
- How to receive completion code information using the D-series enhanced interface

Creating a High-PIN Process

This subsection describes how you can use TACL to run a process at a high PIN or at a low PIN.

You can specify high PIN or low PIN either:

- By setting the #HIGHPIN built-in variable to provide the default value for the RUN command or #NEWPROCESS built-in function
- As a parameter of the RUN command or #NEWPROCESS built-in function to affect only the process you are creating, overriding the default value

Notes

As well as selecting a high PIN at the TACL level, the HIGHPIN object-file attribute for the process you are creating must also be set if the new process is to run at a high PIN. You can set this attribute either at compile time or bind time. See the section that corresponds to the appropriate programming language (Sections 3 through 6) for details.

TACL always ignores the inherent force-low characteristic. See Appendix C, “System Compatibility,” for a discussion of the inherent force-low characteristic.

Setting High PIN as the Default Value

The TACL #HIGHPIN built-in variable provides the default value when you do not specify the HIGHPIN option in a RUN command or #NEWPROCESS built-in function. The default setting for the #HIGHPIN variable is ON; however, you can set this variable using any of the following TACL commands:

```
SET HIGHPIN { ON | OFF }

SET VARIABLE #HIGHPIN { ON | OFF }

#SET #HIGHPIN { ON | OFF }
```

To determine the current value of the #HIGHPIN variable, use the TACL SHOW command:

```
11> SHOW HIGHPIN
```

Setting High PIN for a New Process

You might create a new process (for example, your requester) using the TACL RUN command or #NEWPROCESS built-in function:

```
9>  SET HIGHPIN ON
10> RUN requestr / CPU 3, NAME $REQ, NOWAIT /
```

In the above example, TACL creates the $REC process at a high PIN because the #HIGHPIN built-in variable is set ON. If #HIGHPIN is OFF, you need to specify #HIGHPIN ON in the RUN command line:

```
9>  SET HIGHPIN OFF
10> RUN requestr / HIGHPIN ON, CPU 3, NAME $REQ, NOWAIT /
```

When you issue a D-series RUN command, TACL calls the PROCESS_CREATE_ procedure to create the new process. If the RUN command is unsuccessful, TACL sets the 4-word built-in variable #ERRORNUMBERS to these values:

word [0]	1149
word [1]	PROCESS_CREATE_ *error* returned value
word [2]	PROCESS_CREATE_ *error-detail* parameter
word [3]	0 (zero)

Note that these values are different from the values returned by TACL running under the C-series operating system. C-series TACL returns 1101 in word 0 and NEWPROCESS error information in words 1 and 2. You might need to alter your program if your program controls flow based on these values.

To run a converted process at a low PIN, set the HIGHPIN run option to OFF when you issue the RUN command. For example, the following RUN command causes TACL to create the process at a low PIN in CPU 3 (if a low PIN is available):

```
10> RUN requestr / HIGHPIN OFF, CPU 3, NAME $REQ, NOWAIT /
```

Receiving Completion Codes

Batch processes return completion information indicating whether the process stopped normally or abnormally. D-series TACL uses a new :_COMPLETION^PROCDEATH structure for completion information while providing C-series compatibility by continuing to support the use of :_COMPLETION.

In previous releases, TACL saves Stop (-5) and Abend (-6) messages in the variable :_COMPLETION, if it exists. TACL defines :_COMPLETION as a STRUCT when you log on and the STRUCT remains unless you remove it with a POP command.

D-series TACL receives process deletion messages (-101) instead of Stop and Abend messages. TACL saves each Process deletion message in the variable :_COMPLETION^PROCDEATH, if it exists. TACL defines :_COMPLETION^PROCDEATH as a STRUCT when you log on. The STRUCT remains unless you remove it using the POP command.

Convert your program to reference the :_COMPLETION^PROCDEATH variable instead of :_COMPLETION to make use of the process deletion message information. The structure of :_COMPLETION^PROCDEATH is as follows:

```
[#DEF :_completion^procdeath STRUCT
  BEGIN
     INT        z^msgnumber;
     STRUCT     z^base
                 REDEFINES z^msgnumber;
     BEGIN
        CHAR       byte(0:1);
     END;
     PHANDLE z^phandle;
     INT4    z^cputime;
     INT        z^jobin;
     INT        z^completion^code;
     INT        z^termination^code;
     INT        z^killer^craid
                 REDEFINES z^termination code;
     SSID    z^subsystem;
     PHANDLE z^killer;
     INT        z^termtext^len;
     STRUCT     z^procname;
     BEGIN
        INT           zoffset;
        INT           zlen;
     END;
     INT        z^flags;
     INT        z^reserved(0:2);
     STRUCT     z^data;
     BEGIN
        CHAR       bytes(0:111);
     END;
     STRUCT     z^termtext
                 REDEFINES z^data;
     BEGIN
        CHAR       bytes(0:111);
     END;
     STRUCT     z^procname^
                 REDEFINES z^data;
     BEGIN
        CHAR       bytes(82:193);
     END;
  END;
]
```

For C-series compatibility, you can continue to use :_COMPLETION. If this variable exists, then TACL converts each process deletion system message into a C-series-compatible Stop or Abend message and stores the message in :_COMPLETION. Note, however, that if the message represents an unnamed high-PIN process, the message will not fit in :_COMPLETION and TACL fills :_COMPLETION with zeros.

Using TACL Built-in Functions

This subsection describes variances between the effect of TACL built-in functions on a C-series system and the effect of the same built-in functions on a D-series system. Specifically, the following variances in the D-series TACL language might affect your TACL programs:

- ☐ The #STOP built-in function can return additional error numbers when issued with the ERROR option
- ☐ The #NEWPROCESS and #PROCESS built-in functions return a node name with a process file name or CPU, PIN only in certain circumstances.

Checking the Error When Stopping a Process

Your TACL program might stop a process using the #STOP built-in function and check the result for a file-system error:

```
#SET error [#STOP /ERROR/ $proc]
[#IF error <> 0 |THEN|
  == handle error here
]
```

The D-series TACL process returns value 638 or 639 if the process has been queued for stopping but not actually stopped. If you do not consider these results to be errors, then you should convert your code appropriately:

```
#SET error [#STOP /ERROR/ $proc]
[#IF error <> 0 |THEN|
    == don't consider it an error if the process has been
    == queued for stopping:
    [#IF error <> 638 or error <> 639 |THEN|
    == handle error here
    ]
]
```

Returning a Node Name From #NEWPROCESS or #PROCESS

Your TACL program might use the #NEWPROCESS or #PROCESS built-in functions and expect a process file name or CPU,PIN to be returned with a node name in front of it. In the D-series TACL language:

- ☐ For #NEWPROCESS, the node name is returned only if the process is created on a remote node and the process was started in a NOWAIT manner.
- ☐ For #PROCESS, the node name is returned only if the default process is remote, or if the current current defaults specify a remote node name.

You might need to convert your TACL program to allow for the absence of the node name in all other cases.

Obtaining Lock Information

Your existing program might use the #LOCKINFO built-in function to obtain information about record locks and file locks. For example:

```
#LOCKINFO lock^spec tag buffer
```

Convert your program to use the #FILEGETLOCKINFO built-in function to get information about a lock (either held or pending). Each time you use #FILEGETLOCKINFO, it returns information about one lock. If you need information about several locks, invoke #FILEGETLOCKINFO several times.

In addition to providing the name of the file, you must also provide the following STRUCTs:

`control` A 10-word variable that controls a series of calls to #FILEGETLOCKINFO. Set `control` word [0] to 0 the first time you call #FILEGETLOCKINFO. On successive calls, simply use the value of `control` returned by the previous call.

`lockdesc` Receives the lock information.

`participants` Receives process handles or transaction IDs for processes or transactions that wait for the lock.

In the example below, #FILEGETLOCKINFO returns information about a lock on a disk file.

```
[#DEF control STRUCT
  BEGIN
     INT x(0:9);
  END;
]

[#DEF lockdesc STRUCT
  BEGIN
     INT     lock^type;        == 0 = file, 1 = record
     UINT    flags;            == <0> set if generic lock
     INT     n^participants;   == number of holders/waiters
                               == for lock
     INT     key^length;       == for key-sequenced record
                               == locks; 0 if not
                               == key-sequenced
    CHAR     key(0:255);       == key for key-sequenced
                               == record locks
  END;
]
```

```
[#DEF participants STRUCT
  BEGIN
     STRUCT  locker(0:mp-1);   == mp = max participants
     BEGIN
        UINT    flags;         == <0>    set for process
                               ==        clear for
                               ==        transaction
                               == <1:3>  0 = waiting,
                               ==        1 = granted
                               == <4>    internal use
                               == <5:15> reserved
        FILLER  2;             == reserved
        PHANDLE process;       == process holding or waiting
                               == for the lock
        TRANSID transid        == transaction holding or
          REDEFINES process;   == waiting for the lock
     END;
  END;
]

#SET control 0
#FILEGETLOCKINFO myfile control lockdesc participants
```

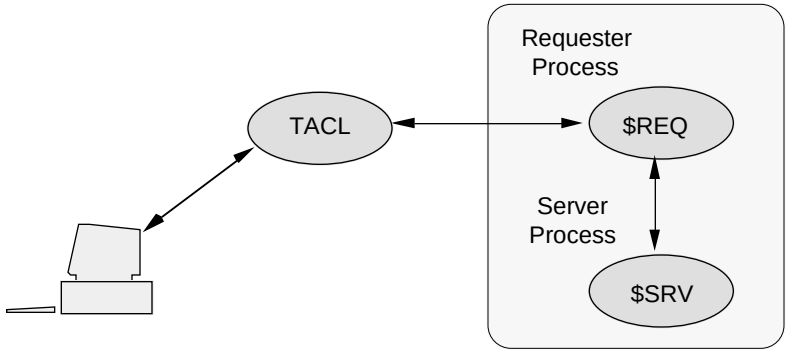
8 Converting Other Parts of an Application

This section describes how to convert the parts of a TAL, COBOL85, C, Pascal, or TACL application that are not described in Sections 3 through 7. The topics in this section are:

- Converting applications that call file-system procedures to manage disk files, including:
  - Manipulating and editing disk file names
  - Maintaining disk files and volumes
- Converting applications that use terminal I/O operations, including a command-interpreter interface and BREAK key handling
- Converting applications that call sequential I/O (SIO) procedures
- Converting Distributed Systems Management (DSM) applications that use the Event Management Service (EMS) or the Subsystem Programmatic Interface (SPI)
- Improving performance using direct transfers for I/O operations
- Converting memory-management procedure calls that allocate, deallocate, and get information about extended data segments

The box in Figure 8-1 contains the processes involved in converting an application. The topics in this section might apply to any of the processes shown in the box for a TAL, COBOL85, C, or Pascal application.

Figure 8-1. Converting Other Parts of an Application



Managing Your Disk Files

This subsection describes how to convert an application that calls Guardian file-system procedures to:

- ☐ Manipulate and edit disk file names
- ☐ Manage disk files and volumes

Converting an application to use D-series file-system procedures is usually optional. However, you might want to convert it because:

- ☐ D-series file-system procedures accept and return file-name string parameters rather than internal-format file-name parameters. You are not required to convert the name from external to internal format before you call a D-series procedure.
- ☐ D-series file-system procedures automatically expand a partially qualified file name to a fully qualified file name using the current settings, including the node name, from the =_DEFAULTS DEFINE VOLUME attribute. You are not required to expand the name before you call a D-series procedure.
- ☐ D-series file-system procedures allow access to remote disk files on other D-series systems in a network with eight-character volume names (seven alphanumeric characters after the dollar sign). C-series procedures can access remote files with volume names that have a maximum of six characters after the dollar sign.
- ☐ D-series file-system procedures return a file-system error value rather than a condition code, which not only makes error checking easier but also simplifies calling file-system procedures from C programs.
- ☐ The D-series FILE_GETINFOLIST_ and FILE_GETINFOLISTBYNAME_ procedures can return more information about a file than the superseded C-series procedures can return.

Manipulating and Editing Disk File Names

The D-series operating system provides new file-system procedures to manipulate disk file names, including:

- ☐ Expanding partially qualified disk file names
- ☐ Extracting and modifying parts of a file name
- ☐ Comparing two file names
- ☐ Using wild-card characters in a file name
- ☐ Upshifting ASCII strings
- ☐ Converting file names between C-series and D-series formats

These topics are described in the following subsections.

Expanding Partially Qualified File Names

Your existing program might call the FNAMEEXPAND or FNAME32EXPAND procedure to expand a partially qualified external file name to a 12-word internal-format file name:

```
length := FNAMEEXPAND (external^filename,
                       internal^filename,
                       default^vol^subvol);
```

Convert your program to call the FILENAME_SCAN_ and FILENAME_RESOLVE_ procedures. FILENAME_SCAN_ checks the syntax of an input file-name string and returns the length in bytes of the file name (provided a valid name is found). FILENAME_RESOLVE_ then converts a partially qualified file name to a fully qualified file name.

Note If you are passing a file name to a D-series procedure, you do not have to call FILENAME_RESOLVE_ after you call FILENAME_SCAN_. A D-series procedure automatically expands a partially qualified file name to a fully qualified file name using the current settings, including the node name, from your =_DEFAULTS DEFINE VOLUME attribute.

In the example below, FILENAME_SCAN_ scans the *name* parameter for a valid file name and, if a valid name is found, returns the length of the name in the *name^byte^count* parameter.

FILENAME_RESOLVE_ then expands the file name into the *file^name* parameter and sets the *file^name^length* parameter to the length in bytes of the fully qualified name:

```
! Check for a valid file name.

error := FILENAME_SCAN_(name:name^length,
                        name^byte^count);
IF NOT error THEN
   BEGIN

   ! Expand the file name.

   error := FILENAME_RESOLVE_(name:name^byte^count,
                              file^name:max^length,
                              file^name^length);
   END;
```

Extracting Parts of a File Name

Your existing program might extract parts from an internal-format file name. For example, you might extract the volume name from a file name.

Convert your program to call the FILENAME_DECOMPOSE_ procedure, which returns one or more parts of a file-name string. You specify the parts that are to be returned with the *level* input parameter. In this example, FILENAME_DECOMPOSE_ returns the volume name and its length. The *level* parameter is set to ZSYS^VAL^FNAME^LEVEL^DEVICE to specify the volume:

```
level := ZSYS^VAL^FNAME^LEVEL^DEVICE;  ! Value = 0.

error := FILENAME_DECOMPOSE_(file^name:file^name^length,
                             volume^name:max^length,
                             volume^name^length,
                             level);
```

Modifying Parts of a File Name

Your existing program might modify parts of an internal-format file name. For example, you might change the subvolume name to a new subvolume name without changing the other parts of the name.

Convert your program to call the FILENAME_EDIT_ procedure, which allows you to modify one or more parts of a file-name string. You specify the parts that are to be modified with the *level* input parameter.

In this example, FILENAME_EDIT_ changes the subvolume name to *subvol^name* and leaves the remaining parts of the file name intact. The *level* parameter is set to ZSYS^VAL^FNAME^LEVEL^SUBVOL to specify the subvolume:

```
level := ZSYS^VAL^FNAME^LEVEL^SUBVOL;  ! Value = 1.

error := FILENAME_EDIT_ (file^name:max^length,
                         file^name^length,
                         subvol^name:subvol^name^length,
                         level);
```

Comparing Two File Names

Your existing program might call the FNAMECOMPARE procedure to compare two file names or device names:

```
error  := FNAMECOMPARE (filename1, filename2);
```

Convert your program to call the FILENAME_COMPARE_ procedure, which compares two file-name strings to determine whether they refer to the same file or device.

FILENAME_COMPARE_ requires a variable-length string for each of the input file names rather than the 12-word internal-format file names. If the file names are incomplete, FILENAME_COMPARE_ uses the current settings, including the node name, from the =_DEFAULTS DEFINE for the unspecified parts.

Each FILENAME_COMPARE_ file-name parameter must be followed by a colon and an integer value, which specifies the length in bytes of the file name. In this example, FILENAME_COMPARE_ compares two file names. The *error* value is then checked for the results:

```
! Compare the two file names and check the results.

error := FILENAME_COMPARE_(file^name1:file^name1^length,
                           file^name2:file^name2^length);
CASE (error) OF
   BEGIN

-1 -> ...                ! The names are different.

 0 -> ...                ! The names are the same.

OTHERWISE -> ...         ! A file-system error occurred
   END;
```

Using Wild-Card Characters in a File Name

Your existing program might use wild-card characters in a file name to specify a set of disk files rather than a single file. The D-series operating system allows the following wild-card characters in all parts of a file name in procedures such as FILENAME_FINDNEXT_ and FILENAME_MATCH_ :

* * Matches zero or more letters, digits, dollar signs ($), or pound signs (#)
* ? Matches one letter, digit, dollar sign ($), or pound sign (#)

Examples of wild-card characters in D-series name strings are:

File-Name String	Specifies
`*ZSPI*`	All files in the current subvolume with names containing ZSPI
`Z???`	All files in the current subvolume with names that begin with the letter Z and have exactly four characters
`\*.$DATA`	All $DATA disk volumes on all nodes in the network
`??????00`	All files in the current subvolume with names that end with the digits 00 and have exactly eight characters
`\*.*.*.*`	All files on all nodes in the network
`\*`	All nodes in the network

Upshifting ASCII Strings

Your existing program might call the SHIFTSTRING procedure to upshift all alphabetic characters in an ASCII string:

```
CALL SHIFTSTRING (file^name^string,
                  string^length,
                  shift^param);  ! Value = 0 (upshift).
```

You might want to convert your program to call the STRING_UPSHIFT_ procedure. This procedure uses both an input and output parameter for the string, which allows you to preserve the unshifted version of the string. For ASCII strings, the output parameter has the same length as the input parameter. For example:

```
error := STRING_UPSHIFT_(in^string:in^string^length,
                         out^string:max^length);
```

Using Both C-Series and D-Series File Names

If you convert your program, it might still contain file names in the C-series 12-word internal format. To convert file names between the C-series and D-series formats, use the procedures described below.

Use the FILENAME_TO_OLDFILENAME_ procedure to convert a D-series file-name string to a C-series 12-word internal-format file name:

```
STRING .d^name[0:ZSYS^VAL^LEN^FILENAME - 1];

INT .c^name[0:11];

...

error := FILENAME_TO_OLDFILENAME_(d^name:name^length,
                                  c^name);
```

Use the OLDFILENAME_TO_FILENAME_ procedure to convert a C-series 12-word internal-format file name to a D-series file-name string:

```
STRING .d^name[0:ZSYS^VAL^LEN^FILENAME - 1];

INT .c^name[0:11];

...

error := OLDFILENAME_TO_FILENAME_(c^name,
                                  d^name:max^length,
                                  name^length);
```

The converted D-series name is always fully qualified (including the node name).

Maintaining Disk Files and Volumes

The D-series operating system provides new file-system procedures to maintain disk files and volumes, including:

- ☐ Creating, renaming, and purging disk files
- ☐ Refreshing a disk volume
- ☐ Getting information about disk files and volumes

These topics are described in the following paragraphs.

Creating a New Disk File

Your existing program might call the CREATE procedure to create a new permanent or temporary disk file:

```
CALL CREATE (disk^file,
             primary^ext^size,
             file^code,
             secondary^ext^size,
             file^type,
             record^length);
```

Convert your program to call the FILE_CREATE_ or FILE_CREATELIST_ procedure to create a permanent or temporary disk file.

FILE_CREATE_ and FILE_CREATELIST_ require a string for the *filename* parameter rather than the 12-word internal-format file name.

For a permanent file, the input *filename* parameter must contain the new file name. If the name is incomplete, both procedures use the current settings, including the node name, from the =_DEFAULTS DEFINE for the unspecified parts.

For a temporary file, the input *filename* parameter contains the name of the volume on which the system creates the temporary file. The system returns the length of the new temporary file name in a separate integer parameter.

FILE_CREATELIST_ also accepts an array of file attributes and values to specify characteristics for a file. For example, you can set alternate-key characteristics for an alternate-key file at the time of creation. The ZSYSDDL file contains LITERAL declarations that you can use with the FILE_CREATELIST_ parameters, including the array of file attributes. To use these declarations, include the appropriate file (ZSYSTAL, ZSYSCOB, ZSYSC, or ZSYSPAS) with your source code file.

In this example, FILE_CREATE_ creates a file named *new^file*:

```
error := FILE_CREATE_(new^file:max^length,
                      new^file^length,
                      file^code,
                      primary^ext^size,
                      secondary^ext^size,
                      max^extents,
                      file^type);
```

Renaming a Disk File

Your existing program might call the RENAME procedure to rename an open disk file:

```
CALL RENAME (file^number,
             new^name);
```

Convert your program to call the FILE_RENAME_ procedure to rename an open disk file. If the file is temporary, FILE_RENAME_ causes the file to become permanent. You must have purge access to the file; otherwise, a security violation (file-system error 48) occurs.

FILE_RENAME_ requires a string for the new file name rather than the 12-word internal-format file name. The length of the name is specified by a separate integer parameter. If the file name is incomplete, FILE_RENAME_ uses the current settings, including the node name, from the =_DEFAULTS DEFINE for the unspecified parts.

In this example, FILE_RENAME_ renames an open file:

```
error := FILE_RENAME_(file^number,
                      new^file^name:new^file^name^length);
```

Purging a Disk File

Your existing program might call the PURGE procedure to delete a closed disk file:

```
CALL PURGE (file^name);
```

Convert your program to call the FILE_PURGE_ procedure to delete the file. FILE_PURGE_ deletes the disk file name from the volume's directory and makes any disk space allocated to the file available to other files.

FILE_PURGE_ requires a string for the file name rather than the 12-word internal-format file name. The length of the name is specified by a separate integer parameter. If the file name is incomplete, FILE_PURGE_ uses the current settings, including the node name, from the =_DEFAULTS DEFINE for the unspecified parts.

To delete a file, you must have purge access to the file; otherwise, a security violation (file-system error 48) occurs. In this example, FILE_PURGE_ deletes a closed file:

```
error := FILE_PURGE_(file^name:file^name^length);
```

Refreshing a Disk Volume

Your existing program might call the REFRESH procedure to refresh a disk volume:

```
CALL REFRESH (volume^name);
```

Convert your program to call the DISK_REFRESH_ procedure. DISK_REFRESH_ writes control information from the file control blocks (FCBs) to the disk volume. It should be used only before the disk volume is brought down (for example, immediately before a cold load).

DISK_REFRESH_ requires a string for the disk name rather than the 12-word internal-format name. The length of the name is specified by a separate integer parameter. In this example, DISK_REFRESH_ refreshes the disk volume indicated by the *volume^name* parameter:

```
error := DISK_REFRESH_(volume^name:volume^name^length);
```

Getting Information About a Disk Volume or File

Your existing program might call one of these procedures to get information about a disk file or volume:

DEVICEINFO	FILEINFO
DEVICEINFO2	FILEINQUIRE
DISKINFO	FILERECINFO

Convert your program to call the FILE_GETINFO_ , FILE_GETINFOLIST_ , FILE_GETINFOBYNAME_ , or FILE_GETINFOLISTBYNAME_ procedure.

FILE_GETINFO_ and FILE_GETINFOLIST_ accept a file number of an open file, while FILE_GETINFOBYNAME_ and FILE_GETINFOLISTBYNAME_ accept a file name to identify the file.

To specify the information that FILE_GETINFOLIST_ or FILE_GETINFOLISTBYNAME_ returns, you set item codes in an *item^list* array. Each integer item code represents a file characteristic. Both procedures return the corresponding file information in a *results* array in the same order specified in the *item^list* array.

The ZSYSDDL file contains LITERAL declarations that you can use with the *item^list* array. To use these declarations, include the appropriate file (ZSYSTAL, ZSYSCOB, ZSYSC, or ZSYSPAS) with your source code file.

If you call FILE_GETINFOBYNAME_ in a waited manner, the system returns the information in the procedure output parameters. However, if you call this procedure in a nowait manner, the system returns the information in system message -108 (nowait FILE_GETINFOBYNAME_ completion). Refer to the *Guardian Procedure Errors and Messages Manual* for the description and format of this message.

In the example below, FILE_GETINFO_ returns the last file-system error for a terminal I/O operation. Then, FILE_GETINFOLISTBYNAME_ uses the file name to return information about a disk file.

```
! Return the last file-system error.

error := FILE_GETINFO_(terminal^file^number,
                       last^terminal^file^error);

...

! Return information for the file indicated by file^name.

error := FILE_GETINFOLISTBYNAME_(file^name:file^name^length,
                                 item^list,
                                 number^of^items,
                                 results,
                                 maximum^results,
                                 results^length,
                                 error^item);
```

Getting Lock Information About a Disk File

Your existing program might call the LOCKINFO procedure to get information about a lock (either held or pending) on a disk volume, disk file, process, or TMF transaction:

```
error := LOCKINFO (search^id,
                   search^type,
                   control^words,
                   buffer^size,
                   buffer);
```

Convert your program to call the FILE_GETLOCKINFO_ procedure to get information about a lock (either held or pending). Each FILE_GETLOCKINFO_ call returns information about one lock. To get information about all locks for an object, make repeated calls to FILE_GETLOCKINFO_.

FILE_GETLOCKINFO_ requires a string for the name rather than the 12-word internal-format name. If the name specifies a disk volume or disk file, FILE_GETLOCKINFO_ uses the current settings, including the node name, from the =_DEFAULTS DEFINE for any unspecified parts. However, if the name specifies a process or TMF TRANSID, the name must include a volume name.

FILE_GETLOCKINFO_ returns information about the lock in the *lock^description* buffer in this format:

word[0]	Lock type (0 = file, 1 = record)	
word[1]	Flags:	.<0> Lock is generic
		.<1:15> Reserved
word[2]	Number of participants	
word[3:4]	Record ID (if the lock is a record type and the file is not key-sequenced)	
word[5]	Key length: Number of bytes in the keys for key-sequenced record locks, or zero for non-key-sequenced record locks	
word[6:*n*]	Locked-key value (where *n* is determined by the length of the buffer)	

The *participants* output parameter of FILE_GETLOCKINFO_ describes the processes (or TRANSIDs) that hold or wait for the lock. The total size of *participants* depends on the *max^participants* parameter. Each 12-word element in *participants* describes one process or TRANSID in this format:

word[0]	Flags:	.<0> Participant (0 = TRANSID, 1 = process handle)
		.<1:3> Grant state (0 = waiting, 1 = granted)
		.<4> Intent flag (0 = lock is not internally set by DP2, 1 = lock is internally set by DP2)
		.<5:15> Reserved
word[1]	Reserved	
word[2:11]	Process handle if word [0].<0> is 1; otherwise, a TRANSID	

The ZSYSDDL file contains LITERAL declarations that you can use with the *lock^description* and *participants* parameters. To use these declarations, include the appropriate file (ZSYSTAL, ZSYSCOB, ZSYSC, or ZSYSPAS) with your source code file.

In the example below, FILE_GETLOCKINFO_ obtains information about a lock on a disk file. The 10-word *control* parameter is used on a series of FILE_GETLOCKINFO_ calls. On the first call, set *control* to zero; on successive calls, return the unchanged value from the previous call.

```
! Return lock information for a disk file.

error := FILE_GETLOCKINFO_(  file^name:file^name^length,
                           ! process^handle ! ,
                           ! trans^id ! ,
                             control,
                             lock^description,
                             lock^desc^length,
                             participants,
                             max^participants);
```

Getting Open Information About a Disk File

Your existing program might call the OPENINFO procedure to get information about the opens for a disk file or device:

```
error := OPENINFO (search^name,
                   previous^tag,
                   primary^process^id,
                   backup^process^id,
                   access^mode,
                   exclusion^mode,
                   sync^depth,
                   file^name,
                   paid,
                   valid^info^mask);
```

Convert your program to call the FILE_GETOPENINFO_ procedure.

FILE_GETOPENINFO_ requires a string rather than the 12-word internal-format name for the input file name or device name parameter. If a file name is incomplete, FILE_GETOPENINFO_ uses the current settings, including the node name, from the =_DEFAULTS DEFINE for the unspecified parts.

FILE_GETOPENINFO_ returns a process handle for the primary and backup opener processes. If a backup opener process does not exist, the backup process handle contains a null value (a -1 in each word).

When you are searching for all opens on a disk volume or for all subdevices for a device, FILE_GETOPENINFO_ returns the output *file^name* parameter. The *file^name* is a variable-length string rather than a 12-word internal-format file name.

In this example, FILE_GETOPENINFO_ obtains information about an open for a disk file. The *search^name* parameter contains the name of the file:

```
! Return information about the last open for a disk file.

error := FILE_GETOPENINFO_(search^name:search^name^length,
                           previous^tag,
                           primary^process^handle,
                           backup^process^handle,
                           access^mode,
                           exclusion^mode,
                           sync^depth,
                           ! file^name:max^buffer^size ! ,
                           ! file^name^length          ! ,
                           paid,
                           valid^info^mask);
```

Using Terminal I/O Operations

If your existing program uses terminal I/O, it might include:

- ☐ A command-interpreter interface
- ☐ BREAK key handling

These topics are described in the following subsections.

Converting a Command-Interpreter Interface

A command-interpreter interface might involve accepting and displaying items such as a CPU value and PIN value or a file name. You might need to modify your existing program to accept and display D-series items (for example, a field for a PIN value for a high-PIN process). The considerations in this subsection also apply to a program that generates printed reports containing D-series items.

Accepting and Displaying CPU and PIN Values

Your existing program might accept, display, or print a three-digit PIN value.

Convert your program to accept, display, or print five-digit PIN values (or six digits if you represent PIN values in octal). Define any PIN variables as integers (or five-digit data items). If you have an error-checking routine for the PIN value, allow PINs with a maximum value of 65535 (or the maximum value allowed for your system).

If your CPU variables or fields hold two or more digits, no changes are necessary. However, if you define a single integer variable for both the CPU and PIN values, redefine each item as a separate integer variable.

Accepting and Displaying Network File and Device Names

Your existing program might accept, display, or print a remote file name or the name of a remote I/O device.

A converted program can access remote files with eight-character volume names or I/O devices with eight-character device names (seven characters after the dollar sign) on other D-series systems in a network. Convert your program to accept, display, or print eight-character remote volume or device names.

Accepting and Displaying Network Process Names

Your existing program might accept, display, or print a remote process name.

A converted program can access remote processes with six-character names (five characters after the dollar sign) on other D-series systems in a network. Convert your program to accept, display, or print six-character remote process names.

Using Sequence Numbers

Usually, a command interpreter operates on the current instance of a process, which is represented by a CPU value and PIN value or by a process name. Tandem does not recommend that you convert your program to accept, display, or print sequence numbers. However, if you must accept, display, or print sequence numbers, allow a maximum of 13 digits for each number.

Displaying File-System Error Numbers

Your existing program might display or print three-digit file-system error numbers.

A D-series file-system error number is an integer variable, which allows a maximum of five digits. Convert your program to display or print five-digit error numbers (or six digits if you display the numbers in octal).

Avoiding Subvolume Defaulting for Disk Files

Your existing program might use subvolume defaulting to represent a disk file name in the form:

```
volume.fileid
```

The D-series operating system does not support subvolume defaulting for disk files. The defaulting scheme for partially qualified file names that are passed to Guardian procedures does not resolve this form of a file name. Also, you cannot use the FILENAME_RESOLVE_ procedure to resolve a name in this form.

Avoid subvolume defaulting in your program. If a disk file name requires the volume name, it must also include the subvolume name.

Accepting a Process String From a Terminal

If your existing program accepts a process string in an input buffer entered at a terminal, you might want to use the PROCESSSTRING_SCAN_ procedure to scan the buffer to extract the process string. In this example, PROCESSSTRING_SCAN_ scans the input buffer named *buffer* and returns a process string in *string^name* and the length of the string in *string^name^length*:

```
error := PROCESSSTRING_SCAN_(buffer:buffer^length,
                             string^length,
                             process^handle,
                             string^type,
                             string^name:max^length,
                             string^name^length,
                             cpu,
                             pin);
```

The format for *string^name* is:

```
[\node-name.]{ name     }
             { cpu,pin  }
```

PROCESSSTRING_SCAN_ includes the node (or system) name if it was entered in the input buffer. If requested, PROCESSSTRING_SCAN_ also returns other items such as the process handle and the CPU and PIN values (if they exist).

Displaying Information About a Process

Your existing program might display or print information about a process using a process ID to identify the process.

In D-series procedures, the process handle replaces the process ID. However, a process handle is not suitable to display or print. To convert a process handle to a process string that is suitable to display or print, use the PROCESSHANDLE_TO_STRING_ procedure:

```
error := PROCESSHANDLE_TO_STRING_(process^handle,
                                  string:max^length,
                                  string^length);
```

In the previous example, PROCESSHANDLE_TO_STRING_ returns a process string in the *string* parameter. The format of the string is:

```
[\node-name.]{ name     }
             { cpu,pin  }
```

The process string includes the node (or system) name only if the process is running on a remote node. You can also use the PROCESSHANDLE_DECOMPOSE_ procedure to return the individual parts of a process handle to display or print. For example:

```
error := PROCESSHANDLE_DECOMPOSE_
                      (process^handle,
                       cpu,
                       pin,
                       system^number,
                       system^name:max^sn^len,
                       system^name^length,
                       process^name:max^pn^length,
                       process^name^length,
                       seq^no);
```

Getting Information About a Process

Your existing program might call one of these procedures to get information about one or more processes:

CREATORACCESSID	MYTERM
GETCPCBINFO	PRIORITY
GETCRTPID	PROCESSFILESECURITY
GETREMOTECRTPID	PROCESSINFO
LOOKUPPROCESSNAME	PROCESSTIME
MOM or MYGMOM	PROGRAMFILENAME

For information about converting these procedures, refer to “Getting Information About a High-PIN Process” in Section 3, “Converting TAL Applications.”

Converting BREAK Key Handling

BREAK key handling might involve taking ownership of the BREAK key and sending and receiving system message -20 (Break).

Taking BREAK Key Ownership

If your existing program calls the SETMODE 11 procedure to take BREAK key ownership, do not use the MYPID procedure to set *parameter-1*. for SETMODE 11. Instead, set *parameter-1* to any positive value. For more information about converting a SETMODE 11 procedure call, refer to the subsection about converting a program to run at a high PIN in the respective section for the language you are using (Sections 3 through 6).

Receiving the Break-on-Device System Message

If your existing program issues the SETMODE 11 call and the BREAK key is pressed, the system sends C-series system message -20 (Break) to your program. Convert your program to receive the D-series system message -105 (Break-on-device). The format of this message is:

sysmsg[0]	-105
sysmsg[1]	File number of the receiver's open file for the terminal that indicated the break, or -1 if the file number is unavailable (for example, from a C-series system)
sysmsg[2] FOR 2	Break tag value specified with a SETPARAM (if used)

Sending a Break-on-Device Message to a High-PIN Process

Your existing program might send a Break-on-device system message to a process using the SENDBREAKMESSAGE procedure:

```
error := SENDBREAKMESSAGE (process^id,
                           break^tag);
```

To send a Break-on-device system message to a high-PIN process, convert your program to use the BREAKMESSAGE_SEND_ procedure.

BREAKMESSAGE_SEND_ requires a process handle rather than a process ID to identify the process. The *receiver^file^number* parameter is the file number in the process that indicates which of the files in the process generated the break. Usually, a program saves this number when it opens the process.

An example of the BREAKMESSAGE_SEND_ procedure is:

```
error := BREAKMESSAGE_SEND_(process^handle,
                            receiver^file^number,
                            break^tag);
```

Using Sequential I/O (SIO) Procedures

An existing program that calls Sequential Input/Output (SIO) procedures can run under the D-series operating system without any changes. However, you must convert specific SIO procedure calls and declarations in a program if:

- ☐ The program uses SIO procedures to read D-series system messages from $RECEIVE
- ☐ The program identifies high-PIN process openers using process handles rather than process IDs

The D-series SIO procedures include these changes:

- ☐ There are no new SIO procedures; the SET^FILE, CHECK^FILE, and OPEN^FILE procedures are modified to support D-series features.
- ☐ The GPLDEFS file contains the D-series SIO declarations, including LITERAL and DEFINE declarations.
- ☐ The file control block (FCB) is 20 words larger and contains new fields for primary and backup process handles.

The conversion required for applications that call SIO procedures is described in the following subsections. For a description of each SIO procedure, refer to the *Guardian Procedure Calls Reference Manual*.

Using the GPLDEFS File

Your existing program should copy the $SYSTEM.SYSTEM.GPLDEFS file, which contains TAL DEFINE and LITERAL declarations required by the SIO procedures. Make sure that your program copies a D-series version of the GPLDEFS file. This file contains a new section called FCB^DEFS^D00, which has these additions:

- ☐ The LITERAL declaration for the size of a D-series FCB is FCBSIZE^D00. Each D-series FCB, including the common FCB, uses 80 words of user data space (a C-series FCB is 60 words).
- ☐ The DEFINE declaration for allocating the common FCB is ALLOCATE^CBS^D00.
- ☐ The DEFINE declaration for allocating the FCB for each file is ALLOCATE^FCB^D00.
- ☐ The DEFINE and LITERAL declarations for the SET^FILE, CHECK^FILE, and OPEN^FILE procedures are revised as described in the following subsections.

Allocating FCBs Using the INITIALIZER

The D-series INITIALIZER procedure supports creator processes that are running at a high PIN. If your existing program calls the INITIALIZER only to read its STARTUP, ASSIGN, or PARAM messages and not to set up its required FCBs, you do not need to make any changes. However, if your program calls the INITIALIZER to allocate its required FCBs, convert the program as follows.

Your existing program might call the INITIALIZER to allocate the run-unit control block (CBS) and the common FCB using the ALLOCATE^CBS DEFINE. In this example, ALLOCATE^CBS allocates a run-unit control block for three I/O files:

```
ALLOCATE^CBS (rucb,       ! Run-unit control block.
              common^fcb, ! Common FCB.
              3);         ! Number of files.
```

Convert your program to allocate the run-unit control block (CBS) and the common FCB using the ALLOCATE^CBS^D00 DEFINE:

```
ALLOCATE^CBS^D00 (rucb,       ! Run-unit control block.
                  common^fcb, ! Common FCB.
                  3);         ! Number of files.
```

Your existing program might call the INITIALIZER to allocate an FCB for $RECEIVE:

```
ALLOCATE^FCB (rec^file,
            "$RECEIVE               ");
```

If you want your process to receive D-series system messages then you should use the ALLOCATE^FCB^D00 DEFINE instead if the ALLOCATE^FCB DEFINE, otherwise your process will receive C-series system messages:

```
ALLOCATE^FCB^D00 (rec^file,
                "$RECEIVE               ");
```

You do not need to change the DEFINE for other files. It does no harm to use ALLOCATE^FCB^D00 for other files except to use additional space.

Allocating FCBs Using Declarations

Your existing program might allocate the common FCB and $RECEIVE FCB using the FCBSIZE LITERAL declaration (60 words). This example shows the allocation of the common FCB, $RECEIVE FCB, and a file FCB for an input file:

```
INT .common^fcb   [ 0:FCBSIZE - 1 ], ! Common FCB.
    .receive^file [ 0:FCBSIZE - 1 ], ! $RECEIVE FCB.
    .input^file   [ 0:FCBSIZE - 1 ]; ! File FCB.
```

Convert your program to allocate the common FCB and $RECEIVE FCB using the FCBSIZE^D00 LITERAL (80 words):

```
INT .common^fcb   [ 0:FCBSIZE^D00 - 1 ], ! Common FCB.
    .receive^file [ 0:FCBSIZE^D00 - 1 ], ! $RECEIVE FCB.
    .input^file   [ 0:FCBSIZE - 1 ];     ! File FCB.
```

Initializing the Common FCB Using SET^FILE

Your existing program might initialize the common FCB as shown in the following example:

```
common^fcb := 0;
```

To initialize a D-series common FCB, you need to convert your program to use the INIT^FILEFCB^D00 parameter in the SET^FILE procedure call as follows:

```
error := SET^FILE (common^fcb, INIT^FILEFCB^D00);
```

Also, make sure that you allocate the common FCB using the FCBSIZE^D00 LITERAL (80 words) as shown above under “Allocating FCBs Using Declarations.”

Initializing a New FCB Using SET^FILE

Your existing program might initialize a new FCB using the SET^FILE procedure and the INIT^FILEFCB parameter. In this example, SET^FILE initializes a new FCB for $RECEIVE:

```
error := SET^FILE (receive^file, INIT^FILEFCB);
```

Convert your program to use the INIT^FILEFCB^D00 parameter:

```
error := SET^FILE (receive^file, INIT^FILEFCB^D00);
```

Specifying an Opener for $RECEIVE Using the SET^FILE Procedure

Your existing program might specify an allowable opener for $RECEIVE (and therefore a process that is allowed to send you messages) using the SET^FILE procedure and the SET^OPENERSPID parameter:

```
error := SET^FILE (receive^file,
                   SET^OPENERSPID,
                   @mom^pid);
```

Convert your program to use the SET^OPENERSPHANDLE parameter in the SET^FILE call. The system then uses a process handle rather than a process ID to identify the allowable opener. In this example, SET^FILE sets the opener's process-handle address for the $RECEIVE FCB to *mom^phandle*:

```
error := SET^FILE (receive^file,
                   SET^OPENERSPHANDLE,
                   @mom^phandle);
```

Specifying System Messages Using the SET^FILE Procedure

Your existing program might specify the system messages it wants to read from $RECEIVE using the SET^FILE procedure and the SET^SYSTEMMESSAGES or SET^SYSTEMMESSAGESMANY parameter.

Setting a bit in the one-word SET^SYSTEMMESSAGES *sys-msg-mask* parameter or the four-word SET^SYSTEMMESSAGESMANY *sys-msg-mask-words* parameter specifies a system message that a program can read:

```
error := SET^FILE (receive^file,
                   SET^SYSTEMMESSAGES,
                   receive^msg^mask,
                   old^msg^mask);
```

The SET^SYSTEMMESSAGES *sys-msg-mask* parameter has the new bit values shown in Table 8-1. The SET^SYSTEMMESSAGESMANY *sys-msg-mask-words* parameter has the new bit values shown in Table 8-2. The values for the remaining bits for each parameter are listed in the *Guardian Procedure Calls Reference Manual*.

Table 8-1. SET^SYSTEMMESSAGES Parameter

Bit	D-Series System Message	C-Series System Message
sys-msg-mask [0]:		
.<5>	-101 (Process deletion)	-5 (Process deletion: Stop)
.<6>	Unused	-6 (Process deletion: Abend)
.<8>	Unused; see *sys-msg-mask-words* [3].<4:7>	-8 (MONITORNET)

Table 8-2. SET^SYSTEMMESSAGESMANY Parameter

Bit	D-Series System Message	C-Series System Message
sys-msg-mask-words [0]:		
.<5>	-101 (Process deletion)	-5 (Process deletion: Stop)
.<6>	Unused	-6 (Process deletion: Abend
.<8>	Unused; see *sys-msg-mask-words* [3].<4:7>	-8 (MONITORNET)
.<9>	-112 (Job process creation)	Unused
sys-msg-mask-words [1]:		
.<4>	-105 (Break on device)	-20 (Break on device)
.<14>	-103 (Process open)	-30 (Process open)
.<15>	-104 (Process close)	-31 (Process close)
sys-msg-mask-words [2]:		
.<8>	-106 (Device type inquiry)	Unused
sys-msg-mask-words [3]:		
.<0>	-102 (Nowait PROCESS_CREATE_ completion)	Unused
.<1>	-107 (Subordinate name inquiry)	Unused
.<2>	-108 (Nowait FILE_GETINFOBYNAME_ completion)	Unused
.<3>	-109 (Nowait FILENAME_FINDNEXT_ completion)	Unused
.<4>	-110 (Loss of communication with node)	Unused
.<5>	-111 (Establishment of communication with node)	Unused
.<6>	-100 (Remote CPU down)	Unused
.<7>	-113 (Remote CPU up)	Unused

Determining an Opener Using the CHECK^FILE Procedure

Your existing program might obtain the address of an opener's process ID using the CHECK^FILE procedure and the FILE^OPENERSPID^ADDR parameter. For example:

```
@opener^pid^addr := CHECK^FILE (common^fcb,
                                FILE^OPENERSPID^ADDR);
```

Convert your program to call the CHECK^FILE procedure using the FILE^OPENERSPHANDLE^ADDR parameter. CHECK^FILE then returns the address of the process handle rather than the process ID to identify the opener. For example:

```
@opener^phandle^addr := CHECK^FILE (common^fcb,
                                    FILE^OPENERSPHANDLE^ADDR);
```

Opening $RECEIVE to Read System Messages

Your existing program might open $RECEIVE using the OPEN^FILE procedure to read system messages:

```
error := OPEN^FILE (common^fcb,
                    receive^file,
                    ! block^buffer       ! ,
                    ! block^buffer^size  ! ,
                    NOWAIT,              ! flags
                    nowait^io^mask);     ! flags^mask
```

The OPEN^FILE *flags* parameter defines a list of LITERAL declarations that can be used with the *flags^mask* parameter to set file-transfer characteristics. A new LITERAL declaration named OLD^RECEIVE is defined for the *flags*.<0> bit. OLD^RECEIVE applies only to $RECEIVE and is ignored for all other files. The values for OLD^RECEIVE are:

0 (the default)	A program reads D-series system messages from $RECEIVE.
1	A program reads C-series system messages from $RECEIVE.

Therefore, if the *flags*.<0> bit is zero in your OPEN^FILE procedure call, no changes are necessary to read D-series system messages provided you have declared and initialized the common FCB and $RECEIVE FCB using the D-series declarations.

If you declare and initialize the common FCB and $RECEIVE FCB using C-series declarations, then the system sends C-series system messages to $RECEIVE regardless of the setting of the *flags*.<0> bit.

If you use a C-series common FCB and a D-series FCB for $RECEIVE, the system returns the SIOERR^OLDCOMMFCB (536) error when you open $RECEIVE.

To read C-series messages after you have declared and initialized the $RECEIVE FCB using D-series declarations, open $RECEIVE using the OLD^RECEIVE LITERAL:

```
error := OPEN^FILE (common^fcb,
                    receive^file,
                    ! block^buffer       ! ,
                    ! block^buffer^size  ! ,
                    OLD^RECEIVE,
                    old^msg^mask);
```

Converting Distributed Systems Management (DSM) Applications

This subsection describes how to convert Distributed Systems Management (DSM) applications that:

- ☐ Use the Event Management Service (EMS) to receive and interpret event messages from a Tandem subsystem
- ☐ Use EMS to generate and report event messages
- ☐ Use the Subsystem Programmatic Interface (SPI) to communicate with a Tandem subsystem

Note Except for the examples in TAL, the symbolic names in this subsection are in DDL (or COBOL85) format with hyphens (-) as separators. If you are writing a TAL program, substitute a circumflex (^) for each hyphen. If you are writing a C or Pascal program, substitute an underscore (_) for each hyphen.

Using the DSM Definition Files

If you are converting a DSM application, make sure that it copies D-series versions of the appropriate DSM definition files that are required for the source language you are using (TAL = TAL, COB = COBOL85, C = C, PAS = Pascal):

SPI definitions	ZSPITAL, ZSPICOB, ZSPIC, or ZSPIPAS
EMS definitions	ZEMSTAL, ZEMSCOB, ZEMSC, or ZEMSPAS
Data communications definitions	ZCOMTAL, ZCOMCOB, ZCOMC, or ZCOMPAS
Subsystem definitions	Z*sss*TAL, Z*sss*COB, Z*sss*C, or Z*sss*PAS (where *sss* is the specific subsystem abbreviation)

Receiving and Interpreting Event Messages

This subsection describes how to convert a DSM application that receives and interprets event messages from a Tandem subsystem.

If you are converting an application, you also need to refer to the Tandem subsystem manual that describes the specific event messages that your application receives from the subsystem. For example, if your application receives event messages generated by the labeled-tape server ($ZSVR), refer to the *Tandem NonStop Kernel Event Management Programming Manual* for a description of the labeled-tape event messages and the tokens in each message.

You might need to convert a DSM application if it receives event messages from a converted subsystem on a D-series system. The converted subsystem can generate event messages that contain D-series tokens, which are unknown to an unconverted application.

For example, suppose an unconverted application tries to extract the process-ID token, ZEMS-TKN-CRTPID, from an event message generated by a converted subsystem. If the event message describes a high-PIN process, the converted subsystem substitutes the process-descriptor token, ZEMS-TKN-PROC-DESC, for the process-ID token. Thus, the unconverted application does not find a process-ID token and does not try to extract the process-descriptor token.

Defining the Event-Message Buffer

Your DSM application might define a buffer to receive the event message. The D-series event-message header contains the new variable-length process-descriptor token, ZEMS-TKN-PROC-DESC, which makes the D-series header approximately 24 bytes larger than a C-series header. Also, because the process-descriptor token is a variable-length string, a D-series event message has a variable length.

If necessary, modify the event-message buffer in your DSM application to hold a larger variable-length event message.

Processing D-Series Tokens

Table 8-3 shows the D-series EMS tokens. You might need to convert your application to process these new tokens. (The D-series EMS tokens that are specific to a subsystem are described in the subsystem's management programming manual.)

Table 8-3. D-Series Event Management Service (EMS) Tokens

Token Name (ZEMS-TKN-)	Token Type (ZSPI-TYP-)	Description
D00	BOOLEAN	Header token that specifies whether the event message is generated by a D-series system or C-series system. Supported only by EMSGET[TKN].
EVTHDR-VSN	ENUM	Header token containing the event-header version. The value for the initial release is 1.
LDEVNUMBER	INT2	Token used for a logical device number (32 bits).
NODENAME	STRING	Token containing the node (system) name of the subsystem that created the event.
NODENUM	INT2	Header token containing the node (system) number of the subsystem that created the event. Replaces ZEMS-TKN-SYSTEM.
PROC-DESC	STRING	Header token containing the process descriptor of the subsystem that created the event. Replaces ZEMS-TKN-CRTPID.
XLDEVNAME	STRING	Token used for a logical device name.
XSENDERID	STRUCT	Token containing the system number (32 bits), CPU (16 bits), and PIN (16 bits) that created the event. Supported only by EMSGET[TKN].
XSENDERID-PD	STRING	Token containing the process descriptor of the subsystem that created the event. Supported only by EMSGET[TKN].

Handling Superseded C-Series Tokens

Your DSM application might try to extract one or more C-series tokens that are superseded by D-series tokens. Table 8-4 shows the C-series tokens that are superseded by D-series tokens. You might need to modify the parts of your application (including any filters and templates) that refer to these superseded C-series tokens.

Table 8-4. Event Management Service (EMS) Superseded Tokens

C-Series Token (ZEMS-TKN-)	D-Series Token (ZEMS-TKN-)	D-Series Token Description	Header Token
CRTPID	PROC-DESC	Process descriptor	Yes
LDEV	LDEVNUMBER	Logical device number	No
LDEVNAME	XLDEVNAME	Logical device name	No
SENDERID	XSENDERID	CPU, PIN, and system number	No
SENDERID	XSENDERID-PD	Process descriptor	No
SYSTEM	NODENAME	Node (system) name	Yes
SYSTEM	NODENUM	Node (system) number	Yes

Handling Cross-Version Tokens

A D-series event-message header might contain a C-series header token (for example, ZEMS-TKN-CRTPID), if the token can hold the required information. If the C-series token cannot hold the required information, the subsystem omits the token from the header. In this case, if your application tries to extract the token, it will receive the missing token error (ZSPI-ERR-MISTKN). If necessary, modify your application to handle the missing token error.

Whether your application is allowed to read tokens from or write tokens in an event message depends on:

- ☐ Whether your application runs on a C-series system or a D-series system
- ☐ Whether the token is a D-series token or a C-series token
- ☐ Whether the token is part of an event message header
- ☐ Whether the token header is a D-series header or a C-series header

Table 8-5 summarizes the cross-version access restrictions for EMS tokens. The column headers indicate whether you are trying to read or write a token and whether your application is running on a C30 system or a D-series system.

Table 8-5. Cross-Version Access Restrictions for EMS Tokens

	C30 System		D System	
Token to Access	Read	Write	Read	Write
Nonheader token in C-series message [1]	Yes	Yes	Yes	Yes
Nonheader token in D-series message [1]	Yes	Yes	Yes	Yes
C-Series token in D-series header	Yes[2]	Yes	Yes[2]	Yes
C-Series token in C-series header	Yes	Yes	Yes	Yes
D-Series token in D-series header	No	No	Yes	Yes
D-Series token in C-series header	No	No	Yes	No[3]

1 For information about a nonheader token, refer to the management programming manual for the subsystem that generated the event message.

2 If the data does not fit in the token, the token is omitted. The application receives the ZSPI-ERR-MISTKN (-8) error.

3 The application receives the ZSPI-ERR-NOTIMP (-11) error.

Replacing a Process-ID Token

Your DSM application might extract a process-ID token from the event-message buffer (for example, a token with token type ZSPI-TYP-CRTPID).

A Tandem subsystem usually substitutes a process-descriptor token for a process-ID token that represents a high-PIN process. However, if it is necessary to supply also the CPU or PIN for a named process, then the process-descriptor token is not enough. The subsystem must supply either a token for the CPU and PIN, or a process handle token. Convert your application to extract the new tokens.

If your application extracts a process-descriptor token, the data is suitable to display or print. However, data in a process-handle token is not suitable to display or print. If you extract a process-handle token, first use a procedure such as PROCESSHANDLE_TO_FILENAME_ or PROCESSHANDLE_DECOMPOSE_ to convert the token data to individual data items before you display or print them.

In the next TAL example, the PROCESSHANDLE_TO_FILENAME_ procedure converts the process handle specified by *process^handle* to a process name in the *proc^name* parameter. The process name always includes the system name. It also includes the sequence number if the *options*.<15> bit is zero (or if the *options* parameter is omitted).

```
error := PROCESSHANDLE_TO_FILENAME_(process^handle,
                                    proc^name:pn^max^len,
                                    proc^name^length,
                                    options);
```

In this TAL example, the PROCESSHANDLE_DECOMPOSE_ procedure returns the parts of the process handle specified by *process^handle*:

```
error := PROCESSHANDLE_DECOMPOSE_(process^handle,
                                  cpu^number,
                                  pin,
                                  system^number,
                                  system^name:sn^max^len,
                                  system^name^length,
                                  process^name:pn^max^len,
                                  process^name^length,
                                  seq^number);
```

Using EMS Filters

If your DSM application uses filters to select specific event messages, EMS provides the D-series FILENAMECOMPARE, DECOMPOSE, and DECOMPOSEERROR functions to use in your filters.

FILENAMECOMPARE compares two variable-length file-name strings. It returns TRUE if the file names are identical; otherwise, it returns FALSE. In this example, FILENAMECOMPARE compares the process-descriptor token, ZEMS-TKN-PROC-DESC, to a specific process-descriptor value:

```
-- Pass the event if the process descriptor matches.

IF FILENAMECOMPARE (ZEMS^TKN^PROC^DESC, "\west.$yab")
 THEN PASS;
```

DECOMPOSE returns the parts of a 12-word internal-format file name, a file-name string, or a process handle. DECOMPOSEERROR returns the most recent DECOMPOSE file-system error if an error occurred, or it returns zero if an error did not occur. In this example, DECOMPOSE returns the parts of a process descriptor, and then DECOMPOSEERROR checks for an error:

```
-- Pass events generated by processes on system \WEST.

IF DECOMPOSE (ZEMS^TKN^PROC^DESC, SYSTEM NAME) = "\WEST"
   AND DECOMPOSEERROR = 0 THEN PASS;

-- Pass events generated by the process named $ZA10.

IF DECOMPOSE (ZEMS^TKN^PROC^DESC,
              DESTINATION NAME, NAME PART) = "$ZA10"
   AND DECOMPOSEERROR = 0 THEN PASS;
```

For more information about EMS filters, refer to the *Event Management Service (EMS) Manual.*

Generating Event Messages

This subsection describes how to convert a DSM application (or a user-written subsystem) that uses EMS to generate and report event messages.

Calling the EMSINIT Procedure

When your unconverted application calls the C-series EMSINIT procedure to build the event-message header, EMSINIT places the process-ID token, ZEMS-TKN-CRTPID, in the event-message header. The D-series EMSINIT procedure substitutes the process-descriptor token, ZEMS-TKN-PROC-DESC, for ZEMS-TKN-CRTPID. The D-series EMSINIT also places the node (system) name and number of the subsystem reporting the event in the ZEMS-TKN-NODENAME and ZEMS-TKN-NODENUMBER tokens.

A D-series event message is larger than a C-series event message, because the D-series header is approximately 24 bytes larger than the C-series header. A D-series event message also has a variable-length structure, because the process-descriptor token ZEMS-TKN-PROC-DESC in the header is a variable-length string. If necessary, modify the event-message buffer in your application to hold the larger variable-length event message.

Specifying File Names

Your DSM application might define a file-name token as a 12-word internal-format file name. For example, it might define a file-name token using token type ZSPI-TYP-FNAME.

Define a file-name token as a variable-length string using a token type such as ZSPI-TYP-STRING. You can use this token for any kind of file name: disk file name, device file name, or D-series process file name.

Specifying Node (System) Names and Numbers

Your DSM application might include a node (system) name or number token in your event message other than in the header tokens described above.

If possible, use a node-name token rather than a node-number token in your event messages. If you define a node-name token, use a variable-length string token with a token type such as ZSPI-TYP-STRING.

In most cases, an 8-bit token for a node number is sufficient. However, if you define a new node-number token, use a 32-bit token with a token type such as ZSPI-TYP-INT2. Put zeros in the first three bytes and the node number in the last byte of this new token.

Specifying CPU and PIN Values

Your DSM application might include an 8-bit CPU or PIN token or a 16-bit token for both the CPU and PIN values in an event message. For example, you might define a CPU or PIN value using token type ZSPI-TYP-BYTE, or a combined CPU and PIN value using token type ZSPI-TYP-INT, ZSPI-TYP-UINT, or ZSPI-TYP-BYTE-PAIR.

Define separate unsigned integer tokens for the CPU and PIN values using a token type such as ZSPI-TYP-UINT.

Specifying a File-System Error

Your DSM application might include an 8-bit token for a file-system error in an event message. For example, you might define an error token using token type ZSPI-TYP-BYTE.

Define file-system error-number tokens as integers using a token type such as ZSPI-TYP-UINT.

Specifying Sequence Numbers

A sequence number is part of a process handle and of a process descriptor (unless it has been removed) and usually does not need to be stored in a separate token. However, if you define a separate token for a sequence number, use a fixed-point number token with a token type such as ZSPI-TYP-INT4.

Specifying Process Handles and Process Descriptors

Your DSM application might include a process-ID token to identify a process in your event message. For example, you might define a process-ID token using token type ZSPI-TYP-CRTPID, ZSPI-TYP-FNAME, or ZSPI-TYP-FNAME32.

Convert your application to define a process-descriptor token or a process-handle token (or both) to identify a process. Tandem recommends that you do not return a synthetic process ID in an event message.

Define a process-descriptor token as a variable-length string using a token type such as ZSPI-TYP-STRING. If you define a process-handle token, use a token type such as ZSPI-TYP-PHANDLE.

Note Tandem recommends that you use a process-descriptor token because:

- ☐ It is in external format, which makes it suitable to display or print.
- ☐ It contains the node (system) and process name, if it exists. After a process pair terminates, you cannot determine the process name from a process handle.

Specifying Device Numbers and Names

Your DSM application might put a logical device number in an integer token such as ZEMS-TKN-LDEV or in a token with token type ZSPI-TYP-INT or ZSPI-TYP-UINT.

A D-series logical device number requires 32 bits. Place a logical device number in the ZEMS-TKN-LDEVNUMBER token or define another 32-bit token using a token type such as ZSPI-TYP-INT2.

Place a device name in the ZEMS-TKN-XLDEVNAME token or define a variable-length string token using a token type such as ZSPI-TYP-STRING.

Note Logical device numbers are often unreliable (for example, with Dynamic System Configuration). Therefore, whenever possible, use a logical device name rather than a logical device number.

Converting Structured Tokens That Contain Obsolete Fields

Your DSM application might define a structured token that contains an obsolete field (that is, a field that cannot hold the required information). For example, you might define a structured token that contains an 8-bit field for a PIN value. If you convert your application, consider the following choices for an obsolete field.

New Structured Token. Define a new structured token with a new field to replace the obsolete field. If you include the old token in the event message, put data in the obsolete field only if it is meaningful; otherwise, put a null value in the field.

If you cannot put either meaningful data or a null value in the obsolete field, omit the old token from the buffer. If a DSM application tries to extract this missing token, it will receive the missing-token error (ZSPI-ERR-MISTKN).

An extensible structured token cannot contain a variable-length field. Therefore, define a separate variable-length token to hold an item such as a process descriptor.

Separate New Simple Tokens. Define a new simple token to replace the obsolete field. For example, you can replace a process-ID field (token type ZSPI-TYP-CRTPID) with a new simple process-descriptor token. Then, define either a new structured token or new simple tokens for the remaining fields.

Converting DSM Applications That Use SPI

This subsection describes how to convert a DSM application that uses the Subsystem Programmatic Interface (SPI) to communicate with a Tandem subsystem.

You need to refer to the management programming manual for the specific subsystem. For example, if your application communicates with FUP, refer to the *File Utility Program (FUP) Management Programming Manual* for a description of the tokens and error lists for the FUP programmatic commands and responses.

Extracting Simple Tokens From the Response Buffer

Your DSM application might try to extract a superseded C-series simple token from the SPI response buffer.

If the information does not fit into a simple token (for example, a high PIN in an 8-bit token), a Tandem subsystem does not return the token in the response buffer. Instead, the subsystem returns a new simple token to hold the value but does not return an error in ZSPI-TKN-RETCODE.

For example, a subsystem might return a process-descriptor token or process-handle token (or both) to represent a high-PIN process. (A subsystem never returns a synthetic process ID to represent a high-PIN process.)

If your application calls the SSGET procedure to extract a token that has been omitted from the response buffer, it will receive SPI error number -8 (token not found). If necessary, convert your application to handle this error.

Extracting Structured Tokens From the Response Buffer

Your DSM application might try to extract a C-series structured token that contains an obsolete field. If the information does not fit into a field of a structured token, the subsystem returns the token as follows:

- ☐ If the field has a defined null value, the subsystem sets the field to its null value and returns the structured token in the response buffer.
- ☐ If the field does not have a defined null value, the subsystem omits the token from the response buffer.

In either case, the subsystem returns the information in a new field of the structured token or a new simple token; it does not return an error in ZSPI-TKN-RETCODE. Refer to the specific management programming manual to determine the token your application should extract.

Processing File-System Error Lists

Your DSM application might process file-system error lists from a subsystem. A subsystem returns a file-system error list when a file-system error occurs during a procedure call from the subsystem. The subsystem nests the file-system error list within a subsystem error list.

For example, if you send a LOAD programmatic command to FUP, FUP in turn calls the file-system WRITE procedure. If a file-system error occurs for the WRITE procedure call, FUP returns a nested error list in the response buffer. The first error list describes the LOAD command error and the second error list describes the WRITE file-system error.

The D-series file-system error-list tokens are:

ZFIL-TKN-ERRORDETAIL (token-type ZSPI-TYP-STRING)

is a conditional token that a subsystem returns if an error condition contains more information than the Z-ERROR integer field can hold.

ZFIL-TKN-XFILENAME (token-type ZSPI-TYP-STRING)

is a conditional token that a subsystem returns if the procedure specified a file-name string parameter. It contains either a file name (including the node name) or a null value if the system could not return a valid file name (for example, an operation on an unopened file).

An error list can also contain a second ZFIL-TKN-XFILENAME token if a procedure specifies a second file-name string parameter (for example, parameters for the FILENAME_COMPARE_ procedure).

ZFIL-TKN-STATUS (token-type ZSPI-TYP-INT)

is a conditional token that a subsystem returns if it calls the FILE_OPEN_ or FILE_OPEN_CHKPT_ procedure.

For FILE_OPEN_ , it is the value of the file-number parameter.

For FILE_OPEN_CHKPT_ , it is the value of the FILE_OPEN_CHKPT_ status parameter, which can have these values:

0 The system opened the backup process successfully.
1 The system opened the backup process successfully, but a warning occurred.
2 An error occurred when the system tried to open the backup process.
3 The system could not communicate with the backup process.
4 The system tried to open the backup process, but an error occurred for the primary process.

The D-series file-system error lists are shown in Table 8-6. For additional information about error lists and extracting tokens from them, refer to the *Subsystem Programmatic Interface (SPI) Programming Manual.*

Table 8-6. D-Series File-System Error Lists

Error Number	ZSPI-TKN-PROC-ERR Value (ZFIL-VAL-)	Name of the Procedure That Returned a Nonzero Value
65	FILE-OPEN-CHKPT	FILE_OPEN_CHKPT_
66	FILE-CREATELIST	FILE_CREATELIST_
67	FILE-OPEN	FILE_OPEN_
68	FILE-PURGE	FILE_PURGE_
69	FILE-CLOSE	FILE_CLOSE_
70	FILE-GETINFOBYNAME	FILE_GETINFOBYNAME_
71	FILE-GETRECEIVEINFO	FILE_GETRECEIVEINFO_
72	FILENAME-COMPARE	FILENAME_COMPARE_
73	FILE-GETOPENINFO	FILE_GETOPENINFO_
74	DISK-REFRESH	DISK_REFRESH_
75	FILE-RENAME	FILE_RENAME_
76	FILENAME-FINDSTART	FILENAME_FINDSTART_
77	FILENAME-FINDNEXT	FILENAME_FINDNEXT_
78	FILENAME-FINDFINISH	FILENAME_FINDFINISH_
80	FILE-CREATE	FILE_CREATE_

Improving I/O Performance Using Direct I/O Transfers

The D-series operating system can improve the performance of I/O operations by transferring data directly between a program's I/O buffers and the I/O device without first allocating an intermediate buffer in the program's process file segment (PFS). A C-series operating system I/O operation uses an intermediate buffer in a program's PFS to transfer data, except when using the SETMODE 141 function (enable or disable large transfers for disk files).

By default, the D-series operating system uses direct transfers for I/O operations. However, the SETMODE 72 function allows a program to control whether direct I/O transfers or intermediate PFS buffers are used for I/O operations.

Using Direct I/O Transfers

Your existing program might use PFS buffers in I/O operations for a file. To convert your program to use direct I/O transfers for a file, open the file using the FILE_OPEN_ procedure. The FILE_OPEN_ default mode for I/O operations is direct transfers.

You can also open the file using the OPEN procedure and then execute a SETMODE 72 function as described in the following subsection.

Using the SETMODE 72 Function

Regardless of the open procedure you use, you can set the I/O transfer mode using the SETMODE 72 function. The SETMODE 72 parameters are:

parameter-1
- 1 = Force the system to use an intermediate buffer in the PFS for I/O transfers for this file.
- 0 = Allow the system to make direct I/O transfers to and from user buffers for this file.

parameter-2 Is not used and should be zero if it is supplied.

If you open a file using the OPEN procedure but need the performance improvement of direct I/O transfers, execute a SETMODE 72 as follows:

```
LITERAL force^pfs^buffers = 72,
        direct^transfers  = 0;
...

CALL SETMODE (file^number,
              force^pfs^buffers,
              direct^transfers);
```

Conversely, if you open a file using the FILE_OPEN_ procedure but need to use PFS buffers for the I/O transfer, execute a SETMODE 72 as follows:

```
LITERAL force^pfs^buffers = 72,
        use^pfs^buffers   = 1;
...

CALL SETMODE (file^number,
              force^pfs^buffers,
              use^pfs^buffers);
```

When You Must Use PFS Buffers

Your existing program might execute one or more of the following types of nowait I/O operations. In each case, you must avoid using direct I/O transfers.

- ☐ Multiple nowait READ[X] operations

  If you execute a second nowait READ[X] before you call AWAITIO[X] to complete the previous nowait READ[X] and you are using the same I/O buffer, use PFS buffers; otherwise, your data might be overwritten.

- ☐ Multiple nowait WRITE[X] operations

  If you initiate more than one nowait WRITE[X] using the same I/O buffer and you change the contents of the buffer between initiation and completion of a WRITE[X] operation, you might write incorrect data. Whether you are using PFS buffers or direct transfers, avoid changing the contents of your WRITE[X] buffer between initiation and completion of the WRITE[X] operation.

- ☐ Multiple nowait WRITEREAD[X] operations

  If multiple WRITEREAD[X] operations share the same I/O buffer, use PFS buffers and restore the buffer to the same value immediately after each WRITEREAD[X] finishes.

For the above cases, if you open a file using the OPEN procedure, no conversion is necessary. When you execute the OPEN procedure, the system uses PFS buffers by default for nowait I/O operations.

However, if you open a file using the FILE_OPEN_ procedure, issue a SETMODE 72 function with *parameter-1* set to 1 to force the use of PFS buffers. Refer to "Using the SETMODE 72 Function" in the previous subsection for more information.

Converting Memory-Management Procedure Calls

The D-series operating system provides new memory-management procedures to allocate, make accessible, deallocate, and get information about extended data segments. It also provides a new procedure for checking the address limits of either your process's data segment or an extended data segment. These new procedures are described in the following subsections.

Note Converting an application to use the D-series memory-management procedures is necessary only if the application shares an extended data segment with a high-PIN process using the PIN method. Converting to use the SEGMENT_USE_ and ADDRESS_DELIMIT_ procedures is optional at this time but will be required under future versions of the operating system.

Allocating an Extended Data Segment

Your existing program might call the ALLOCATESEGMENT procedure to allocate an extended data segment:

```
error := ALLOCATESEGMENT (segment^id,
                          segment^size,
                          swap^file^name);
```

Convert your program to call the SEGMENT_ALLOCATE_ procedure. If you specify a swap file, use a variable-length string file name rather than the 12-word internal file name. Specify the length in bytes of the swap file name as a separate integer parameter.

In the following example, SEGMENT_ALLOCATE_ allows the calling process to share the extended data segment identified by *segment^id* using the PIN method. The system returns the base address of the segment in *base^address*:

```
error := SEGMENT_ALLOCATE_(segment^id,
                           segment^size,
                           ! swap^file^name:length ! ,
                           error^detail,
                           ! pin ! ,
                           ! segment^type ! ,
                           @base^address);
```

To allocate an extended data segment for a backup process, call the SEGMENT_ALLOCATE_CHKPT_ procedure from the primary process. This procedure supersedes the CHECKALLOCATESEGMENT procedure.

Making an Extended Data Segment Accessible

Your existing program that calls the ALLOCATESEGMENT procedure probably calls the USESEGMENT procedure to make the segment current and therefore accessible to your application:

```
old^segment^id := USESEGMENT (segment^id);
```

You can optionally convert your program to call the SEGMENT_USE_ procedure. In this example, the ID of the extended segment that was accessible before this call to SEGMENT_USE_ is returned in the output parameter *old^segment^id*. If no extended segment was accessible before this call, -1 is returned.

```
error := SEGMENT_USE_ (segment^id,
                       old^segment^id,
                       ! base^address ! ,
                       error^detail);
```

Deallocating an Extended Data Segment

Your existing program might call the DEALLOCATESEGMENT procedure to deallocate an extended data segment:

```
CALL DEALLOCATESEGMENT (segment^id);
```

Convert your program to call the SEGMENT_DEALLOCATE_ procedure. The *flags* integer parameter specifies whether dirty pages are written to the swap file. (A dirty page is a page in memory that has been updated but not written to the swap file.) The *flags*.<0:14> bits must be zero; *flags*.<15> can be:

0 Dirty pages are written to the swap file (the default action).
1 Dirty pages are not written to the swap file.

In this example, SEGMENT_DEALLOCATE_ deallocates the extended data segment identified by *segment^id*. The procedure uses the default *flags* value:

```
error := SEGMENT_DEALLOCATE_(segment^id,
                             ! flags ! ,
                             error^detail);
```

To deallocate an extended data segment for a backup process, call the SEGMENT_DEALLOCATE_CHKPT_ procedure from the primary process. This procedure supersedes the CHECKDEALLOCATESEGMENT procedure.

Getting Information About an Extended Data Segment

Your existing program might call the SEGMENTSIZE procedure to get information about a currently allocated extended data segment:

```
segment^size := SEGMENTSIZE (segment^id);
```

Convert your program to call the SEGMENT_GETINFO_ procedure, which returns this information:

- ☐ The size of the segment in bytes
- ☐ The swap file name associated with the segment
- ☐ The length in bytes of the swap file name
- ☐ The base address of the segment

In this example, SEGMENT_GETINFO_ returns information about the extended data segment identified by *segment^id*:

```
error := SEGMENT_GETINFO_(segment^id,
                          segment^size,
                          swap^file^name:max^length,
                          swap^file^length,
                          error^detail,
                          base^address);
```

To return the same information about an extended data segment that is currently allocated by the backup process of a process pair, call the SEGMENT_GETBACKUPINFO_ procedure from the primary process.

Extended Segment Size

On both TNS and TNS/R systems, the maximum size of an extended segment is 127.5 megabytes as of the C30.06 release. The SEGMENT_ALLOCATE_ and RESIZESEGMENT Guardian procedures no longer allocate extended segments larger than 127.5 megabytes.

Change your program if it uses more than the new maximum size. If a program allocates extended segments that are too large, SEGMENT_ALLOCATE_ returns error 5.

Checking Address Limits

Your existing program might call the LASTADDR procedure to obtain the relative address of the last word in your application process's user data segment:

```
last^address := LASTADDR;
```

Or your existing program might call the LASTADDRX procedure to obtain the last relative address available in the specified relative data segment (which must be accessible at the time of the call):

```
last^ext^address := LASTADDRX ( rel^segment^num );
```

You can optionally convert your program to call the ADDRESS_DELIMIT_ procedure. This procedure obtains the addresses of the first and last bytes of a particular area of your logical address space. You can use it to obtain the boundaries of either your process's user data segment or of a currently accessible extended data segment.

You supply an address contained within the address area of interest, passing it to ADDRESS_DELIMIT_ in the value parameter *address*. Optionally, you can use the *segment-id* output parameter to obtain the segment ID of the area if it is an extended data segment. You can also use the *address-descriptor* output parameter to obtain a set of flags that describe the area.

In the following example, the address of a local variable contained in the user data segment is passed to ADDRESS_DELIMIT_. The procedure returns the addresses of the first and last bytes of the user data segment.

This example shows that the output addresses can be assigned either to a simple variable (LOW^ADDR) or to a pointer variable (HIGH^ADDR). After a successful call to ADDRESS_DELIMIT_, HIGH^ADDR designates the last byte of the user data segment.

```
INT      LOCAL^VARIABLE;
INT(32)  LOW^ADDR;
STRING  .EXT HIGH^ADDR;
INT      ERROR,
         ERROR^DETAIL;
           .
           .
           .
ERROR := ADDRESS_DELIMIT_ ($XADR(LOCAL^VARIABLE),
                           LOW^ADDR,
                           @HIGH^ADDR,
                           ! address^descriptor ! ,
                           ! segment^ID ! ,
                           ERROR^DETAIL);

IF ERROR <> 0 THEN CALL ERROR^HANDLER;
```

Handling the Message System Interface

Prior to the D-series versions of the operating system, the RESERVELCBS procedure allowed you to reserve link control blocks (LCBs) for sending and receiving messages. It also performed certain secondary functions concerning the limits on the number of extended memory link control blocks (XLBs).

Under the D-series versions of the operating system, RESERVELCBS performs no function but can still be used without error. Reserving link control blocks is no longer applicable, and the secondary functions of the procedure can be accomplished by calling CONTROLMESSAGESYSTEM.

If your existing program calls RESERVELCBS specifically to reserve LCBs, then you can safely remove the call. Such calls typically specify a number of LCBs significantly greater than 1. For example, the following call reserves 20 receive LCBs and 20 send LCBs:

```
CALL RESERVELCBS(20,            !receive LCBs
                 20);           !send LCBs
```

If your existing program calls RESERVELCBS to raise the limit on the number of XLBs, then you should convert your program to use the CONTROLMESSAGESYSTEM procedure. For such programs, the default limit of 255 XLBs for receive XLBs or 1023 for send XLBs was considered inadequate. Programs typically remove the limit by specifying a 1 for the number of send or receive LCBs. For example, your existing program might remove the limit on receive XLBs as follows:

```
CALL RESERVELCBS(1,             !receive LCBs
                 0);            !send LCBs
```

Replace this call to RESERVELCBS with a call to the CONTROLMESSAGESYSTEM procedure. Choose a value for the limit on XLBs that is suitable for your application. A typical value might be 2000:

```
LITERAL  receive^xlbs^limit = 0;
...
value := 2000;
CALL CONTROLMESSAGESYSTEM(receive^xlbs^limit,
                          value);
```

9 Converting to TNS/R Systems

Most TNS programs written for the C30 and D-series versions of the operating system can run on a TNS/R system without modification. Variances between TNS and TNS/R systems, however, might require modification in some programs, particularly in privileged TAL programs. One variance (see "Odd-Byte References," later in this subsection) applies to C programs as well.

Before you run a TNS object file on a TNS/R system, you can accelerate it (optimize it to take advantage of the TNS/R architecture). Most accelerated object files run faster than nonaccelerated object files. The *Accelerator Manual* tells how to accelerate object files.

This section discusses general considerations about and variances between the TNS and TNS/R systems.

General Considerations

The following considerations apply to all object files (whether accelerated or not) that you are running on a TNS/R system:

- ☐ Extended segment limit checking
- ☐ Overflow results

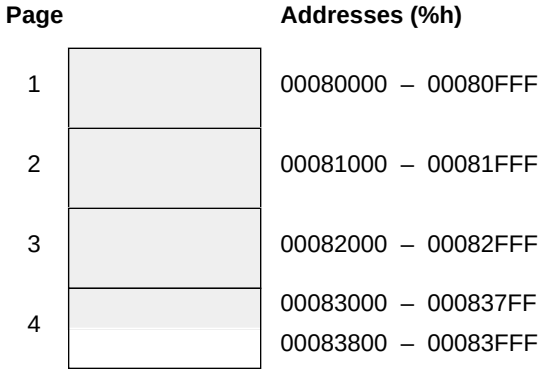
Extended Segment Limit Checking

On both TNS and TNS/R systems, the maximum size of an extended segment is 127.5 megabytes as of the C30.06 release. Extended segments on TNS and TNS/R systems differ as follows:

System	Number of Memory Pages	Size of Memory Page	Address Boundary Checking by the Processor
TNS	64	2048-byte	To a byte boundary
TNS/R	32	4096-byte	To a memory page boundary

Eliminate any references to data beyond the extended segment's logical limit. Addressing past the end of extended segments fails differently on TNS/R systems than on TNS systems. You can notice the difference when debugging addressing errors:

- ☐ On TNS systems, programs that reference data beyond their extended segment's logical limit fail immediately.
- ☐ On TNS/R systems, programs that reference data beyond their extended segment's logical limit might or might not fail. For example, programs that reference data between the logical end of an extended segment and the next 4-KB memory-page boundary continue executing without an exception. That is, a program can reference data located between the end of the segment and the end of the memory page (unshaded area) without an exception. Addresses are shown here in hexadecimal.



Overflow Results

Multiplication and division overflow results are undefined on all Tandem systems. Furthermore, overflow results of integer multiplication and division instructions (except LMPY) are incompatible between the TNS and TNS/R systems. The incompatible instructions are:

```
IDIV     DDIV     QDIV     LDIV
IMPY     DMPY     QMPY
```

Overflow results of addition and subtraction instructions (and LMPY) are compatible on both systems. The compatible instructions are:

```
IADD     DADD     QADD     LMPY
ISUB     DSUB     QSUB
INEG     DNEG     QNEG
```

You must test for overflow if your program expects multiplication or division overflow. Use $OVERFLOW in an IF statement immediately after the operation that might overflow:

```
FIXED(0) count, time, result;

CODE (RDE; ANRI %577; SETE);   !Disable overflow traps
result := count * time;
IF $OVERFLOW THEN              !Test for overflow
   result := 0D;               !Fix the overflow result
```

General-Case Variances

The following variances apply to all object files (whether accelerated or not) that you are running on a TNS/R system:

- ☐ Trap handlers that use the register stack
- ☐ Trap handlers that use the program counter
- ☐ Privileged instructions
- ☐ Nonprivileged references to system global data
- ☐ Stack wrapping
- ☐ Odd-byte references
- ☐ Data swap file size
- ☐ Passing the address of P-relative objects
- ☐ Shift instructions with dynamic shift counts

Trap Handlers That Use the Register Stack

During program execution, the trap mechanism handles all error and exception conditions not related to input or output. User-written trap handlers differ on TNS and TNS/R systems as follows:

- ☐ On TNS systems, trap handlers can be functions and they can change the register stack (registers R0 through R7).
- ☐ On TNS/R systems, the reported register stack contents are imprecise. Trap handlers must not be functions and they can reliably change only certain registers. The trap handlers, however, can:
  - ☐ Print error messages and abend
  - ☐ Clear overflow and resume
  - ☐ Resume after a loop timer
  - ☐ Jump to a restart point by changing the trap variables P, L, ENV, space ID, and S

  Within a trap handler, you can put values into global variables. You can also save the register stack contents (although imprecise) as follows:

```
CODE (PUSH %777);    !Save register stack contents
```

Change your programs to comply with the preceding restrictions. Find trap handlers by looking for calls to the system procedure ARMTRAP with parameters (address of label, address of data area). ARMTRAP specifies an entry point into the program where execution is to begin if a trap occurs. You need not change ARMTRAP (-1,-1) calls that cause programs to abend on traps.

Trap Handlers That Use the Program Counter

The reported program counter register, P, is imprecise for all code. Do not rely on the value of P to determine program flow or to calculate the target of a jump.

You can change the P trap variable to a valid restart point (such as a cleanup point in the outer block). When doing so, you can make changes that are consistent with this to ENV and the space ID. However, do not perform arithmetic on P.

For example, the results of the following operations are undefined:

```
INT trap_p = 'L' - 2;      !Location of the P register
trap_p := trap_p '+' 1;    !Undefined
trap_p := trap_p '-' 1;    !Undefined
```

Change your programs to comply with the preceding restrictions. Find trap handlers by looking for calls to the system procedure ARMTRAP with parameters (address of label, address of data area). ARMTRAP specifies an entry point into the application program where execution is to begin if a trap occurs. You do not need to change ARMTRAP (-1,-1) calls that cause programs to abend on traps.

For more guidelines on writing trap handlers, see "Trap Handlers That Use the Register Stack" earlier in this section.

Privileged Instructions

TOTQ and RCPU are currently nonprivileged instructions on TNS processors; they are privileged instructions on TNS/R processors.

TOTQ Test the out queue
RCPU Read the CPU number

Remove these instructions from nonprivileged programs. On TNS/R systems, nonprivileged programs that use these instructions fail with an Instruction Failure exception.

Nonprivileged References to System Global Data

Only privileged programs can access system global data. When a nonprivileged program references system global data, results differ depending on the system:

- ☐ On TNS processors, the program accesses the user global data segment instead.
- ☐ On TNS/R processors, the program fails with an Instruction Failure exception.

Use a text editor to search for 'SG' in nonprivileged programs and remove references to system global data from nonprivileged programs.

The following nonprivileged procedure references the system data at SG[0].

```
INT cpuno = 'SG' + 0;      !SG equivalencing
INT .EXT xcpuno;           !Extended pointer

PROCEDURE foo;             !Nonprivileged procedure
  BEGIN
  IF cpuno = 1 THEN        !TNS/R system traps
    BEGIN                  !TNS system accesses G[0]
    !Lots of code
    END;
  @xcpuno := $XADR(cpuno);
  IF xcpuno = 1 THEN       !TNS/R system traps
        . . .              !TNS system accesses G[0]
  END;
```

Stack Wrapping

If a nonprivileged program tries to allocate variables past the end of the user data segment, results might differ depending on the system:

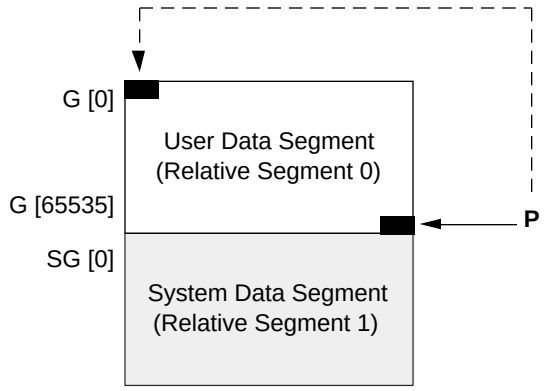
- ☐ On TNS processors, the allocation wraps back to the user data segment.
- ☐ On TNS/R processors, the allocation is unpredictable; it might wrap or trap.

Remove any constructs that result in stack wrapping. Change addressing operations so that they stay within the user data segment.

The following examples show addressing operations that might not wrap back on the stack on TNS/R processors as they do on TNS processors.

```
STRUCT s(*);
  BEGIN
  INT i;
  INT(32) d;
  INT j;
  END;

PROC exam;
  BEGIN
  INT .p (s);
  STRUCT s2(s);
  @p := -1;     !-1 = 65535 (unsigned); structure starts
                ! at the end of the user data segment
  s2 ':=' p FOR $LEN(s2) BYTES;   !On TNS/R systems,
  p.d := %h12345678%d;            ! these operations
  p.j := 6;                       ! might wrap or trap
  END;
```

On TNS processors, structure P starts at the end of the user data segment and wraps back to G[0], as shown by the dotted line in the following figure. On TNS/R processors, structure P might wrap or trap.



902

Odd-Byte References

TNS/R systems do not support the same odd-byte references to doublewords (4-byte units) or quadruplewords (8-byte units) as TNS systems do. Odd-byte references are made when the least significant bit (bit 31) of an extended pointer is set.

- TNS systems ignore bit 31 of an extended pointer and fetch or store data starting from the byte prior to the addressed one to ensure that the address is on a word boundary. This behavior is probably not what you intended.
- On TNS/R systems, odd-byte references to doublewords or quadruplewords are unpredictable; programs might trap or they might continue executing. Extended pointers to variables accessed as doublewords or quadruplewords must not have their least significant bit set. If your program was written following good TAL programming practices, odd-byte references are not a concern.

In TAL, data types stored as doublewords or quadruplewords include INT(32), FIXED(n), INT(64), REAL, and REAL(64). In C, data types stored as doublewords or quadruplewords include long, unsigned long, long long, float, and double.

To correct your program, remove odd-byte references to doublewords or quadruplewords. Also, make sure that all pointers contain addresses. Many odd-byte references to doublewords and quadruplewords are caused by uninitialized pointers. (%37777000000D, the null address, is an appropriate value to use for initializing a pointer until it is given another address.)

The results of the following example are unpredictable on a TNS/R system:

```
REAL(64) .EXT rptr;
STRING .EXT sptr = rptr;    !Same pointer as RPTR
@sptr := @sptr + 1d;
IF sptr = "Z" THEN ...      !OK
IF rptr = 0.0L0 THEN ...    !Results are unpredictable
```

In the following example, an odd-byte alignment error occurs on a TNS/R system because the program refers to a nil pointer. The extended pointer P contains a byte address. When P is set to -1, the structure starts on an odd-byte boundary. The program might trap or it might continue executing.

```
STRUCT s(*);
  BEGIN
  INT i;
  INT(32) d;
  INT j;
  END;

PROC test;
  BEGIN
  INT .EXT p(s);
  @p:= -1d;                 !Nil 32-bit pointer to structure
  p.d := %habcd1234%d;      !Results are unpredictable
  END;
```

Data Swap File Size

On all TNS/R systems, the amount of memory allotted to a process for user data is rounded up to the nearest multiple of 4 KB. Additionally, on D20 TNS/R systems, the data swap file for the process requires 16 disk pages (32 KB) beyond this amount.

For example, if you call the PROCESS_CREATE_ procedure on a TNS system and specify a value of 3 for the *memory-pages* parameter, 6 KB of memory would be allotted to the new process for user data and 6 KB of disk space would be required for the data swap file. But on a D20 TNS/R system, the amount of memory allotted to the new process would be rounded up to 8 KB while an additional 32 KB of disk space would be required for the swap file, making the size of the swap file 40 KB.

When creating a process on a TNS/R system, if you specify the name of an existing disk file as the data swap file, you must ensure that the file has sufficient disk space allocated to satisfy these requirements.

Passing the Addresses of P-Relative Objects

Do not pass the address of a P-relative object to other routines in programs larger than one TNS code segment. Programs access the wrong object or an address fault occurs if you pass a:

- ☐ 16-bit address on TNS systems
- ☐ 16-bit address on TNS/R systems
- ☐ 32-bit extended address on TNS systems

However, if you pass a 32-bit extended address on TNS/R systems, programs access the specified object and continue execution. This is a program error that TNS/R systems do not detect.

Do not write programs that rely on this feature. There is one supported exception to this restriction: you can pass user code addresses into user library, system code, or system library routines.

Look for statements that pass a 32-bit extended address of a P-relative object to other routines. Recode statements that pass a 32-bit extended address of a P-relative object to other routines.

Shift Instructions With Dynamic Shift Counts

The implementation of TNS instructions for signed arithmetic and unsigned logical shifts with dynamic shift counts differs between TNS and TNS/R systems. This difference applies to both single-word (16-bit) and double-word (32-bit) shift instructions.

For single-word shift operations with dynamic shift counts:

- ☐ TNS systems accept counts within the range of 0 to 255. Shift counts of 16 to 255 are treated as 16.
- ☐ TNS/R systems counts within the range of 0 to 31. Shift counts of 16 to 31 are treated as 16. Shift counts greater than 31 give undefined results.

For double-word shift operations with dynamic shift counts:

- ☐ TNS systems accept counts within the range of 0 to 255. Shift counts of 32 to 255 are treated as 32.
- ☐ TNS/R systems accept counts within the range of 0 to 32,767. Shift counts greater than 32 are treated as 32.

Dynamic shift counts that fall outside of the acceptable ranges give undefined results.

The ALS, LLS, ARS, and LRS instructions implement single-word shifts and the DALS, DLLS, DARS, and DLRS instructions implement double-word shifts.

The TAL compiler generates these instructions for the bit-shift operators ('<<', '>>', <<, and >>) if the operand to the right of the operators is not a constant. These instructions can also be found in TAL CODE statements. Refer to the appropriate system description manual for more information about the TNS instruction set. Refer to the *TAL Reference Manual* for details on TAL bit-shift operators.

To correct your program, you must recode statements that use dynamic shift counts greater than 31. Use a text editor to search for “ALS 0”, “LLS 0”, “ARS 0”, and “LRS 0” in TAL CODE statements. Make sure that the operands for these instructions do not use shift counts greater than 31. Also search for the TAL bit-shift operators: unsigned left and right shift ('<<' and '>>') and signed left and right shift (<< and >>). Make sure that in cases where the left operand is a 16-bit unit and the shift count is dynamic, the count is never greater than 31.

In the following example, the value of p is arithmetically shifted to the right 35 positions. The value of differs on TNS and TNS/R systems.

```
INT p, nbits, q;  ! INT variables
p := -128;
nbits := 35;      ! Shift value of 35-bits
q := p >> nbits;  ! On TNS systems,
                  ! same as p >> 16, q = -1
                  ! On TNS/R systems,
                  ! same as p >> (35-32), q = -16
```

Appendix A Guardian Procedures

This appendix lists the D-series Guardian procedures that supersede C-series Guardian procedures. If you are converting an application, find each C-series procedure used in your application in the left-hand column of Table A-1. The corresponding item in the right-hand column shows the D-series procedure used for conversion. A hyphen (-) in the right-hand column means that there is no new D-series procedure and that you should continue to use the C-series procedure.

For a description of each procedure, refer to the *Guardian Procedure Calls Reference Manual*.

Table A-1. Guardian Procedures (Page 1 of 8)

C-Series Procedure	D-Series Procedure Used for Conversion
ABEND	PROCESS_STOP_
ACTIVATEPROCESS	PROCESS_ACTIVATE_
ADDDSTTRANSITION	–
ADDRTOPROCNAME	–
ALLOCATESEGMENT	SEGMENT_ALLOCATE_
ALTER	FILE_ALTERLIST_
ALTERPRIORITY	PROCESS_SETINFO_
ARMTRAP	–
AWAITIO[X]	–
CANCEL	–
CANCELPROCESSTIMEOUT	–
CANCELREQ	–
CANCELTIMEOUT	–
CHANGELIST	–
CHECK^BREAK	–
CHECK^FILE	–
CHECKALLOCATESEGMENT	SEGMENT_ALLOCATE_CHKPT_
CHECKCLOSE	FILE_CLOSE_CHKPT_
CHECKDEALLOCATESEGMENT	SEGMENT_DEALLOCATE_CHKPT_
CHECKDEFINE	–
CHECKMONITOR	–
CHECKOPEN	FILE_OPEN_CHKPT_

Table A-1. Guardian Procedures (Page 2 of 8)

C-Series Procedure	D-Series Procedure Used for Conversion
CHECKPOINT	–
CHECKPOINTMANY	–
CHECKPOINTMANYX	–
CHECKPOINTX	–
CHECKRESIZESEGMENT	–
CHECKSETMODE	–
CHECKSWITCH	–
CLOSE	FILE_CLOSE_
CLOSE^FILE	–
COMPUTEJULIANDAYNO	–
COMPUTETIMESTAMP	–
CONTIME	–
CONTROL	–
CONTROLBUF	–
CONTROLMESSAGESYSTEM	–
CONVERTPROCESSNAME	FILENAME_RESOLVE_
CONVERTPROCESSTIME	–
CONVERTTIMESTAMP	–
CPUTIMES	–
CREATE	FILE_CREATE[LIST]_
CREATEPROCESSNAME	PROCESSNAME_CREATE_
CREATEREMOTENAME	PROCESSNAME_CREATE_
CREATORACCESSID	PROCESS_GETINFO[LIST]_
CURRENTSPACE	–
DAYOFWEEK	–
DEALLOCATESEGMENT	SEGMENT_DEALLOCATE_
DEBUG	–
DEBUGPROCESS	PROCESS_DEBUG_
DEFINEADD	–
DEFINEDELETE	–
DEFINEDELETEALL	–
DEFINEINFO	–

Table A-1. Guardian Procedures (Page 3 of 8)

C-Series Procedure	D-Series Procedure Used for Conversion
DEFINELIST	–
DEFINEMODE	–
DEFINENEXTNAME	–
DEFINEPOOL	–
DEFINEREADATTR	–
DEFINERESTORE	–
DEFINERESTOREWORK[2]	–
DEFINESAVE	–
DEFINESAVEWORK[2]	–
DEFINESETATTR	–
DEFINESETLIKE	–
DEFINEVALIDATEWORK	–
DELAY	–
DEVICEINFO[2]	FILE_GETINFOBYNAME_, DEVICE_GETINFOBYNAME_, or DEVICE_GETINFOBYLDEV_
DISKINFO	FILE_GETINFOLISTBYNAME_
DNUMIN	–
DNUMOUT	–
EDITREAD	–
EDITREADINIT	–
EMSADDBUFFER	–
EMSADDSUBJECT	–
EMSADDSUBJECTMAP	–
EMSADDTOKENMAPS	–
EMSADDTOKENS	–
EMSGET	–
EMSGETTKN	–
EMSINIT[MAP]	–
EMSTEXT	–
FILEERROR	–
FILEINFO	FILE_GETINFO[LIST][BYNAME]_
FILEINQUIRE	FILE_GETINFO[LIST][BYNAME]_
FILERECINFO	FILE_GETINFO[LIST][BYNAME]_

Table A-1. Guardian Procedures (Page 4 of 8)

C-Series Procedure	D-Series Procedure Used for Conversion
FIXSTRING	–
FNAME32COLLAPSE	–
FNAME32EXPAND	FILENAME_SCAN_ and FILENAME_RESOLVE_
FNAME32TOFNAME	–
FNAMECOLLAPSE	–
FNAMECOMPARE	FILENAME_COMPARE_
FNAMEEXPAND	FILENAME_SCAN_ and FILENAME_RESOLVE_
FNAMETOFNAME32	–
FORMATCONVERT[X]	–
FORMATDATA[X]	–
GETCPCBINFO	PROCESS_GETINFOLIST_
GETCRTPID	PROCESS_GETINFO[LIST]_
GETDEVNAME	FILENAME_FINDNEXT_
GETPOOL	–
GETPPDENTRY	PROCESS_GETPAIRINFO_
GETREMOTECRTPID	PROCESS_GETINFO[LIST]_
GETSYNCINFO	–
GETSYSTEMNAME	NODENUMBER_TO_NODENAME_
GETTMPNAME	–
GETTRANSID	–
GIVE^BREAK	–
GROUPIDTOGROUPNAME	–
GROUPNAMETOGROUPID	–
HALTPOLL	–
HEAPSORT	–
INITIALIZER	–
INTERPRETINTERVAL	–
INTERPRETJULIANDAYNO	–
INTERPRETTIMESTAMP	–
JULIANTIMESTAMP	–

Table A-1. Guardian Procedures (Page 5 of 8)

C-Series Procedure	D-Series Procedure Used for Conversion
KEYPOSITION[X]	–
LABELEDTAPESUPPORT	–
LASTADDR[X]	ADDRESS_DELIMIT_
LASTRECEIVE	FILE_GETRECEIVEINFO_
LOCATESYSTEM	NODENAME_TO_NODENUMBER_
LOCKFILE	–
LOCKINFO	FILE_GETLOCKINFO_
LOCKREC	–
LOOKUPPROCESSNAME	PROCESS_GETPAIRINFO_
MESSAGESTATUS	–
MESSAGESYSTEMINFO	–
MOM	PROCESS_GETINFO[LIST]_
MONITORCPUS	–
MONITORNET	–
MONITORNEW	–
MOVEX	–
MYGMOM	PROCESS_GETINFO[LIST]_
MYPID	PROCESS_GETINFO_ and PROCESSHANDLE_DECOMPOSE_
MYPROCESSTIME	–
MYSYSTEMNUMBER	PROCESS_GETINFO_ and PROCESSHANDLE_DECOMPOSE_
MYTERM	PROCESS_GETINFO[LIST]_
NEWPROCESS[NOWAIT]	PROCESS_CREATE_
NEXTFILENAME	FILENAME_FINDNEXT_
NO^ERROR	–
NUMIN	–
NUMOUT	–
OPEN	FILE_OPEN_
OPEN^FILE	–
OPENEDIT	OPENEDIT_
OPENINFO	FILE_GETOPENINFO_

Table A-1. Guardian Procedures (Page 6 of 8)

C-Series Procedure	D-Series Procedure Used for Conversion
POSITION	–
PRIORITY (obtain priority)	PROCESS_GETINFO[LIST]_
PRIORITY (alter priority)	PROCESS_SETINFO_
PROCESSACCESSID	PROCESS_GETINFO[LIST]_
PROCESSFILESECURITY	PROCESS_SETINFO_
PROCESSINFO	PROCESS_GETINFO[LIST]_
PROCESSORSTATUS	–
PROCESSORTYPE	–
PROCESSTIME	PROCESS_GETINFOLIST_
PROGRAMFILENAME	PROCESS_GETINFO[LIST]_
PURGE	FILE_PURGE_
PUTPOOL	–
READ[X]	–
READ^FILE	–
READLOCK[X]	–
READUPDATE[X]	–
READUPDATELOCK[X]	–
RECEIVEINFO	FILE_GETRECEIVEINFO_
REFRESH	DISK_REFRESH_
REMOTEPROCESSORSTATUS	–
REMOTETOSVERSION	–
RENAME	FILE_RENAME_
REPLY[X]	–
REPOSITION	–
RESERVELCBS	–
RESETSYNC	–
RESIZEPOOL	–
RESIZESEGMENT	–

Table A-1. Guardian Procedures (Page 7 of 8)

C-Series Procedure	D-Series Procedure Used for Conversion
SAVEPOSITION	–
SEGMENTSIZE	SEGMENT_GET[BACKUP]INFO_
SENDBREAKMESSAGE	BREAKMESSAGE_SEND_
SET^FILE	–
SETLOOPTIMER	–
SETMODE	–
SETMODENOWAIT	–
SETMYTERM	PROCESS_SETSTRINGINFO_
SETPARAM	–
SETSTOP	–
SETSYNCINFO	–
SETSYSTEMCLOCK	–
SHIFTSTRING	STRING_UPSHIFT_
SIGNALPROCESSTIMEOUT	–
SIGNALTIMEOUT	–
SPI_BUFFER_FORMATFINISH_	–
SPI_BUFFER_FORMATNEXT_	–
SPI_BUFFER_FORMATSTART_	–
SPI_FORMAT_CLOSE_	–
SSGET	–
SSGETTKN	–
SSIDTOTEXT	–
SSINIT	–
SSMOVE	–
SSMOVETKN	–
SSNULL	–
SSPUT	–
SSPUTTKN	–
STEPMOM	PROCESS_SETINFO_
STOP	PROCESS_STOP_
SUSPENDPROCESS	PROCESS_SUSPEND_
SYSTEMENTRYPOINTLABEL	–

Table A-1. Guardian Procedures (Page 8 of 8)

C-Series Procedure	D-Series Procedure Used for Conversion
TAKE^BREAK	–
TEXTTOSSID	–
TIME	–
TIMESTAMP	–
TOSVERSION	–
TRANSIDTOTEXT	–
UNLOCKFILE	–
UNLOCKREC	–
USERDEFAULTS	–
USERIDTOUSERNAME	–
USERNAMETOUSERID	–
USESEGMENT	SEGMENT_USE_
VERIFYUSER	–
WAIT^FILE	–
WRITE[X]	–
WRITE^FILE	–
WRITEREAD[X]	–
WRITEUPDATE[X]	–
WRITEUPDATEUNLOCK[X]	–
XBNDSTEST	–
XSTACKTEST	–

Appendix B System Messages

Table B-1 lists the C-series and corresponding D-series user-level system messages. For the description and format of these system messages, refer to the *Guardian Procedure Errors and Messages Manual*.

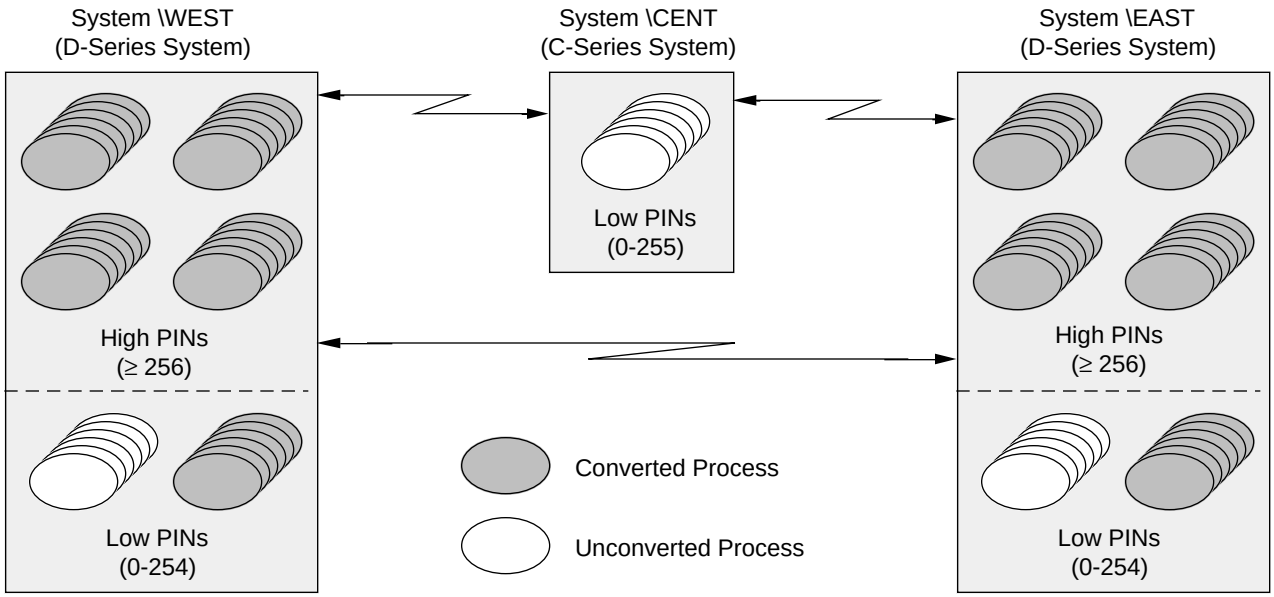
Table B-1. System Messages

C-Series System Message		D-Series System Message	
–2	CPU down: process MONITORCPUS	–2	CPU down: process MONITORCPUS
–2	CPU down: named process deletion	–101	Process deletion: CPU down
–3	CPU up	–3	CPU up
–5	Process deletion: Stop	–101	Process deletion: Stop
–6	Process deletion: Abend	–101	Process deletion: Abend
–8	Change in status of network node	–100	Remote CPU down
–8	"	–110	Loss of communication with node
–8	"	–111	Establishment of communication with node
–8	"	–113	Remote CPU up
–9	Job process creation	–112	Job process creation
–10	SETTIME	–10	SETTIME
–11	Power on	–11	Power on
–12	NEWPROCESSNOWAIT completion	–102	PROCESS_CREATE_ completion
–13	System message buffer overrun	–13	System message buffer overrun
–20	Break on device	–105	Break on device
–21	3270 device status received	–21	3270 device status received
–22	Elapsed time timeout	–22	Elapsed time timeout
–23	Memory lock completion	–23	Memory lock completion
–24	Memory lock failure	–24	Memory lock failure
–26	Process time timeout	–26	Process time timeout
–30	Process open	–103	Process open
–31	Process close	–104	Process close
–32	Process CONTROL	–32	Process CONTROL
–33	Process SETMODE	–33	Process SETMODE
–34	Process RESETSYNC	–34	Process RESETSYNC
–35	Process CONTROLBUF	–35	Process CONTROLBUF
–37	Process SETPARAM	–37	Process SETPARAM
–38	Queued message cancellation	–38	Queued message cancellation
–40	Device type inquiry	–106	Device type inquiry
–41	Nowait DEVICEINFO2 completion	–41	Nowait DEVICEINFO2 completion
None		–107	Subordinate name inquiry
None		–108	Nowait FILE_GETINFOBYNAME_ completion
None		–109	Nowait FILENAME_FINDNEXT_ completion

Appendix C System Compatibility

This appendix describes the compatibility between processes on C-series and D-series systems in a network. A typical network has C-series systems with all processes at low PINs and D-series systems with low-PIN unconverted processes, low-PIN converted processes, and high-PIN converted processes. Figure C-1 shows three systems in a typical network.

Figure C-1. Network of C-Series and D-Series Systems



This appendix describes these compatibility issues for processes on C-series systems and D-series systems in a network:

- ☐ Identifying disks and devices using C-series and D-series file names
- ☐ Identifying processes using C-series and D-series process identifiers
- ☐ Ensuring compatibility between processes running on a C-series system and descendent processes that run on a D-series system
- ☐ Allowing opens by high-PIN requesters of an unconverted process
- ☐ Communicating with a high-PIN process from an unconverted process

These issues do not consider Guardian file and process security or security provided by the Safeguard subsystem. For more information about file and process security, refer to the D-series *Guardian Programmer's Guide.*

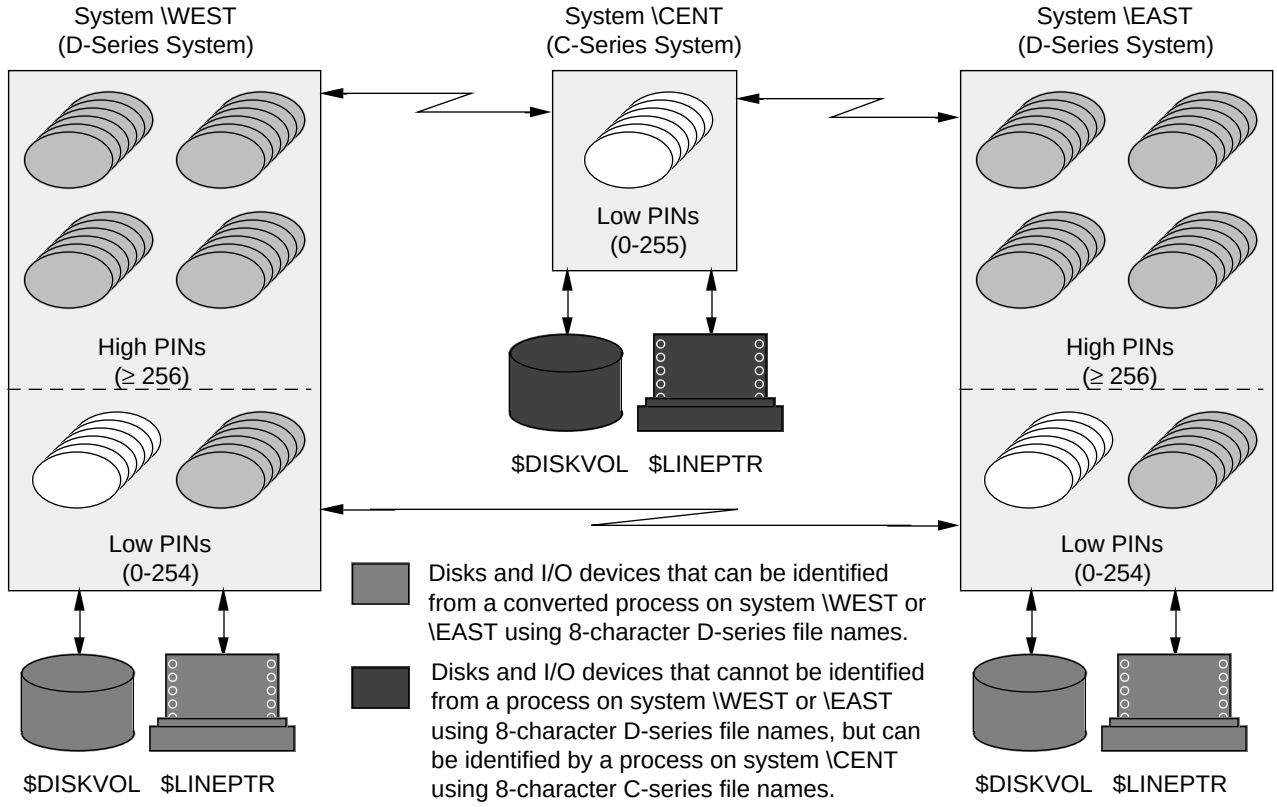
Identifying Disks and I/O Devices

A process uses a file name to identify a disk volume or an I/O device such as a printer or tape drive. An unconverted process can use eight-character file names (one to seven characters after the dollar sign) to identify local disk volumes or I/O devices. However, an unconverted process cannot use an eight-character name to identify remote disk volumes or I/O devices.

Using D-series file names, a converted process on a D-series system can identify local or remote disk volumes or I/O devices with eight-character names if they are on other D-series systems in the network. However, a converted process on a D-series system follows the C-series file-name identification rules when accessing remote C-series systems: it cannot identify remote disk volumes or I/O devices with an eight-character name if they are on C-series systems in the network.

Figure C-2 shows the use of eight-character file names to identify disk volumes and I/O devices in a network.

Figure C-2. Identifying Disk Volumes and I/O Devices



Identifying Processes

The D-series operating system uses D-series process file names and process handles to identify processes. For compatibility with C-series operating systems, the D-series operating system also supports C-series process file names and process IDs. The following paragraphs describe the use of C-series and D-series process identifiers.

Note that the ability to identify a process does not imply that you can open it and communicate with it. For rules about which processes you can and cannot open, see "Allowing Opens by High-PIN Requesters" and "Communicating With a Named High-PIN Process" later in this appendix.

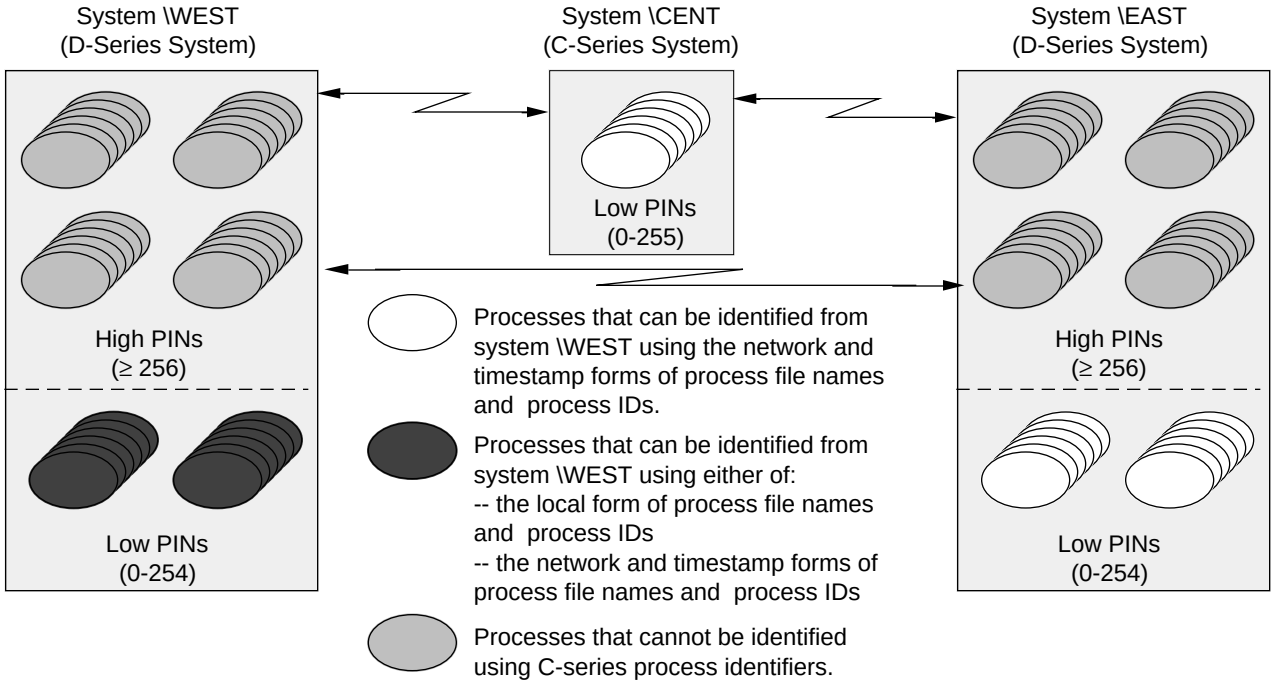
Using C-Series Process Identifiers

When you are identifying processes in a network, a C-series process file name or process ID has certain restrictions. You cannot identify a high-PIN process on either a local or remote D-series system. You can identify a local or remote low-PIN process on a C-series system or D-series system using:

- ☐ The timestamp form of an unnamed C-series process file name or process ID, as long as the PIN is less than 255
- ☐ The network named form of a C-series process file name or process ID, if the process name has fewer than five characters after the dollar sign

Figure C-3 shows the use of C-series process file names and process IDs to identify processes in a network.

Figure C-3. Identifying Processes Using C-Series Process Identifiers



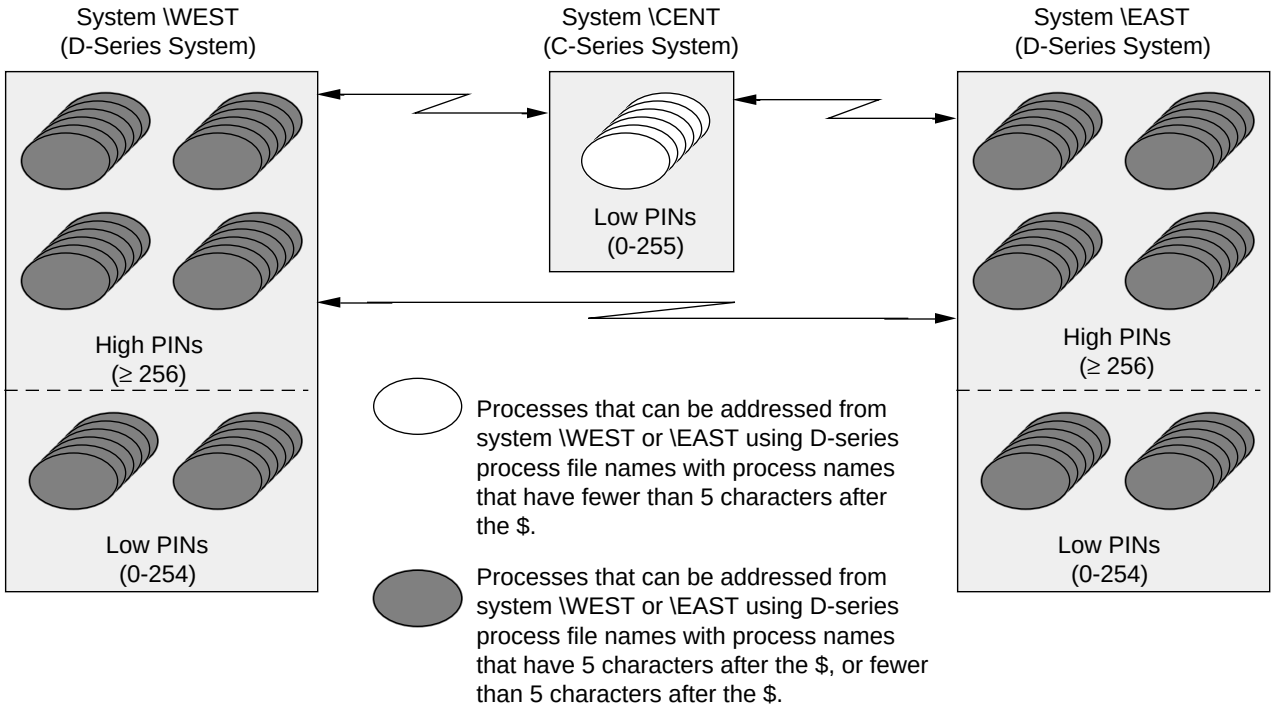
Using D-Series Process File Names

Using a D-series process file name, a converted process can identify a high-PIN or low-PIN process with a name that has up to five characters after the dollar sign (for example, $ZAB22) on D-series systems in a network.

However, a process on a D-series system follows the C-series process-identification rules when accessing remote C-series systems: it cannot identify a remote process with a name that has five characters after the dollar sign on C-series systems in the network.

Figure C-4 shows the use of D-series process file names to identify processes in a network.

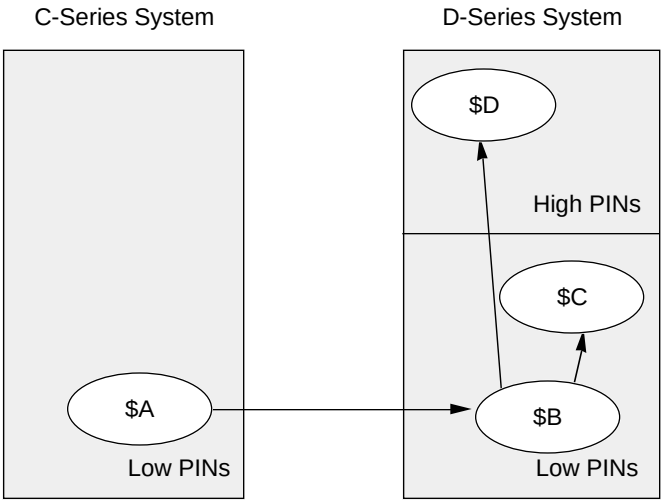
Figure C-4. Identifying Processes Using D-Series Process File Names



Ensuring Compatibility: The Inherited Force-Low Characteristic

Processes that create other processes often need to communicate with their descendent processes. Communication problems can occur when a process running on a C-series system creates a process running on a D-series system that in turn creates additional processes, some of which run at a high PIN. Figure C-5 shows the problem.

Figure C-5. Process Creation Between C-Series and D-Series Systems



Process $B can run only at a low PIN because it is created by a process running on a C-series system and must therefore have been created by the NEWPROCESS procedure. Processes created by $B, however, can run at a high PIN or low PIN, because $B is running on a D-series system and therefore can use the PROCESS_CREATE_ procedure.

Using the Inherited Force-Low Characteristic

To help ensure compatibility, all processes created by process $B run at a low PIN by default. The mechanism used to achieve this default action is the inherited force-low characteristic.

If a process is started on a D-series system by a process on a C-series system, then the new process not only runs at a low PIN but also has its inherited force-low characteristic set. Because this flag normally propagates to all its descendent processes, all descendents of the C-series process normally run at a low PIN.

A process also has its inherited force-low characteristic set if its creator used PROCESS_CREATE_ with *create-options*.<15> set to 1 (the force-low flag), or if it was created using the C-series-compatible NEWPROCESS procedure.

Overriding the Inherited Force-Low Characteristic

If there is no need for a process to communicate with an ancestor process that runs at a low PIN, then you can override the inherited force-low characteristic and allow the new process to run at a high PIN or low PIN, depending on the force-low flag and on whether the HIGHPIN object-file attribute is set.

You use *create-options*.<10> in the PROCESS_CREATE_ procedure to override the inherited force-low characteristic (the ignore force-low flag). The new process does not have its inherited force-low characteristic set.

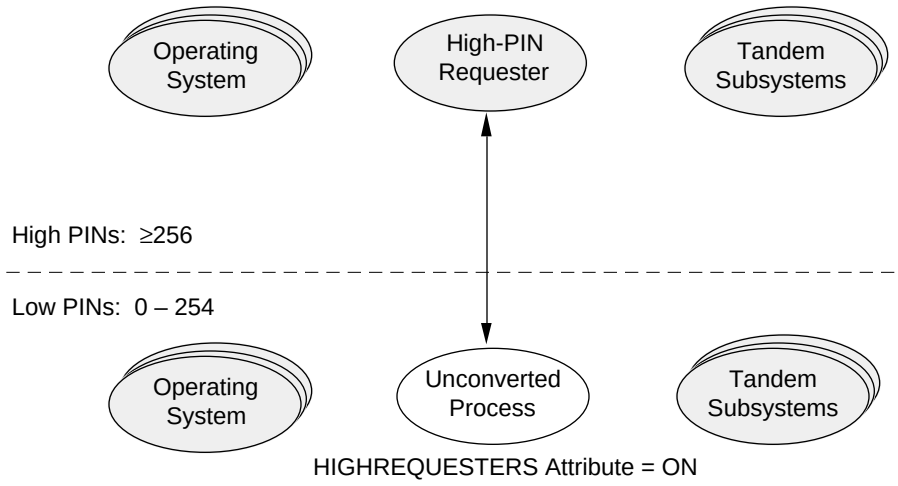
Allowing Opens by High-PIN Requesters

An unconverted process on a D-series system can be opened by a high-PIN requester process and receive requests from the requester process if the unconverted process:

- ☐ Has its HIGHREQUESTERS object-file attribute set
- ☐ Does not examine the identity of its openers or requesters

Figure C-6 shows an unconverted process and a high-PIN requester.

Figure C-6. Allowing Opens by High-PIN Requesters



A high-PIN process cannot open an unconverted process unless the unconverted process has the HIGHREQUESTERS object-file attribute set. If a high-PIN process attempts to open a low-PIN process that does not have this attribute set, the high-PIN process receives file-system error 560. The unconverted process is not affected.

For information about setting the HIGHREQUESTERS object-file attribute, refer to "Setting the HIGHREQUESTERS Attribute to Allow High-PIN Openers" in the respective section for each language (Sections 3 through 6).

Using Synthetic Process IDs

A synthetic process ID is a process name or timestamp followed by the CPU number and a PIN value of 255. A synthetic process ID allows an unconverted server to support high-PIN openers (for example, in an opener table).

If the low-PIN process enables high-PIN requesters with the HIGHREQUESTERS object-file attribute, the system returns a synthetic process ID for these cases:

- ☐ As the output process-ID parameter for a high-PIN process from a RECEIVEINFO or LASTRECEIVE procedure call after the low-PIN process reads a system message from $RECEIVE.
- ☐ As the output process-ID parameter for a high-PIN process from a MOM procedure or the ancestor process ID field of a LOOKUPPROCESSNAME or GETPPDENTRY procedure call.
- ☐ As the process ID of a primary high-PIN process in the C-series system message -30 (Process open) when receiving notification of the backup-process open.
- ☐ As the process-ID for a high PIN primary or backup process identified by the OPENINFO procedure as the owner of an open file.
- ☐ As the process-ID of a high PIN process identified by a LOCKINFO procedure as holding a lock on the specified file.

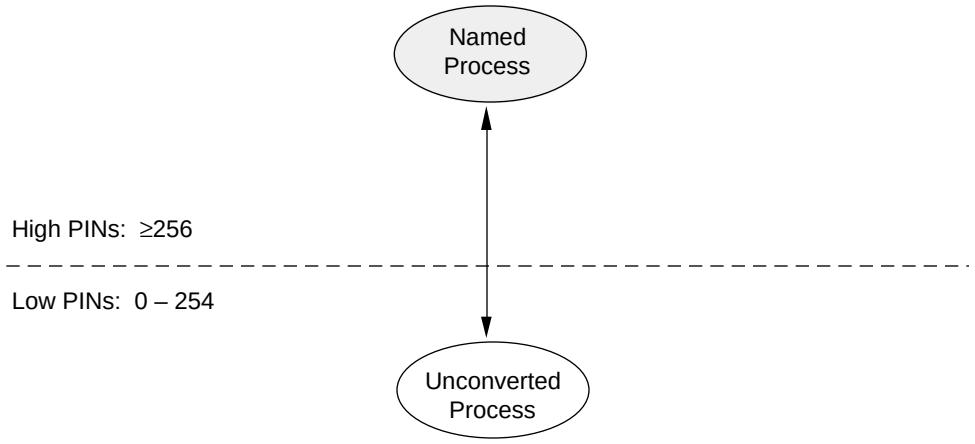
Note Because a synthetic process ID cannot uniquely describe a high-PIN process, Tandem recommends that you do not use them for other cases. For example, do not use a synthetic process ID in C-series procedures such as PROCESSINFO and SENDBREAKMESSAGE, extract information from them, display them, or put them in messages (including event messages).

Communicating With a Named High-PIN Process

An unconverted process running on a D-series system can communicate with a named high-PIN process as described in the following paragraphs. However, to communicate with an unnamed high-PIN process, the process must be converted.

Figure C-7 shows an unconverted process and a named high-PIN process.

Figure C-7. Communicating With a Named High-PIN Process



An unconverted process can open a named high-PIN process using the OPEN procedure. The OPEN procedure uses the process name to determine the process (or process pair) and does not require the CPU and PIN values from the process ID:

```
CALL OPEN (process^file^name,
           file^number);
```

After opening the high-PIN process, the unconverted process can send requests using a procedure such as WRITE[X] or WRITEREAD[X]. These procedures do not require any conversion to send or receive messages from high-PIN processes.

An unconverted process can close a high-PIN process using the CLOSE procedure:

```
CALL CLOSE (file^number);
```

Appendix D Considerations for Migrating Any Application

This appendix provides information about processing changes that may require minor conversion of your application even if you do not intend to take advantage of the D-series extensions. Without these modifications, your C-series application might:

- ☐ Fail to run on a D-series system
- ☐ Produce erroneous or unexpected results when run on a D-series system

Potential Application Problems

Potential application problems on a D-series system can be caused by:

- ☐ Using undocumented procedures
- ☐ Relying on undocumented side effects of documented procedures

Undocumented Procedures

Some undocumented C-series procedures are not supported on D-series systems. The following table lists some of the most commonly used undocumented procedures.

Undocumented Procedure	Documented Procedure Replacement
BULKREAD, BULKWRITE, BULKAWAITIO	READX, WRITEX, AWAITIOX (with SETMODE 141 for transfers of up to 56 KB)
GETPEEKXGLOBAL	PROCESS_GETINFOLIST_
SETDATAFREELIST or SETINDEXFREELIST	None. These procedures apply only to DP1 files. DP1 cannot exist in a D-series system

To ensure that your C-series application runs successfully on a D-series system, modify your application to eliminate the use of undocumented procedures.

Undocumented Side Effects of Documented Procedures

If your application relies on undocumented side effects of documented procedures, it might function differently on your D-series system than it did on your C-series system.

To ensure that your C-series application runs successfully on a D-series system, modify your application to eliminate the use of undocumented side effects.

Other Potential Application Problems

Other potential application problems are summarized in table D-1 and described in detail in subsequent subsections.

Table D-1. Potential Application Problems

Affected application component	Cause of potential problem	Refer to the following subsection for more information
Condition codes	Application checks undefined condition codes	"Undefined Condition Codes Contain Meaningless Information"
DEFINEREADATTR and DEFINEINFO attributes	Application does not account for the system name in file names or does not provide for zero suppression	"Enhanced Attribute Values Returned From DEFINEs"
Device names	Application relies on device names of unacceptable length	"D-Series Systems Must Be Named"
Device simulator process	Application propagates TMF transactions to a device simulator process	"TMF Transactions Not Propagated to Device Simulator Process Automatically"
INITIALIZER procedure	Application estimates FCBs for INITIALIZER procedure	"INITIALIZER Procedure Enhanced"
Nowait write buffer	Application uses nowait write buffers incorrectly	"Nowait Write Buffer Integrity"
Pool space	Application relies on a precisely calculated amount of pool space	"Pool Space Address Adjustment"
Process IDs	Application receives a process ID from a process that runs on a C-series system with a PIN of 255	"For a Process ID of 255 it is Important to Know the Source System"
Process names	Application relies on process names of unacceptable length	"D-Series Systems Must Be Named"
Process pairs	Application (process pair) does not account for CPU down system message (-2)	"System-Message Protocol for Process Pairs Includes CPU Down Message"
SDU buffer	Application uses SDU buffers that are too small	"Aggregate SDU Length Checking Enhanced"
System naming	Application assumes a system can be unnamed	"D-Series Systems Must Be Named"
Temporary file names	Application relies on 4-digit temporary file names	"Temporary File Names Have 7 Digits"
	Application relies on first 4 digits of file names being unique	"Temporary File Names Have 7 Digits"
TERMPROCESS	Application uses TERMPROCESS	"TERMPROCESS Replaced by ATP6100"

INITIALIZER Procedure Enhanced

For D-series systems, the number of FCBs specified in the INITIALIZER procedure must match the number of FCBs actually allocated for the RUCB and common FCB. If the numbers do not match, the INITIALIZER issues the message "INITIALIZER: Invalid format or wrong number of FCBs specified" and terminates.

For C-series systems, specifying the wrong number of FCBs did not cause a processing error.

To ensure that your C-series application runs successfully on a D-series system, if it uses the INITIALIZER procedure, ensure that the number of specified FCBs for the procedure is correct.

Undefined Condition Codes Contain Meaningless Information

For D-series and TNS/R systems, undefined condition codes are set by accidental side effects. A condition code is considered to be undefined for a Guardian procedure if the documentation for the procedure does not explicitly refer to condition code values.

For example, although the NEWPROCESS procedure has never returned a condition code, a program that checks the condition code might have run successfully on a C-series system:

```
CALL NEWPROCESS (file^name, ...);
IF <> THEN ...
```

To ensure that your C-series application runs successfully on a D-series system, modify your application to eliminate checking of undefined condition codes.

Aggregate SDU Length Checking Enhanced

For D-series systems, the TLAM PORT interface performs enhanced length checking on aggregate SDUs and aggregate SDU buffers.

On D-series systems, if the total specified length of incoming aggregate SDUs exceeds the length of an aggregate SDU buffer in your application, the I/O operation is not executed. File-system error 22 (application buffer address out of bounds) is returned to your application.

On C-series systems, if the total specified length of incoming aggregate SDUs exceeded the length of an aggregate SDU buffer in your application, the I/O operation was executed but a truncated SDU was put into the buffer. and no warning was returned to your application.

To ensure that your C-series application runs successfully on a D-series system, if it uses aggregate SDU buffers, ensure that a total aggregate SDU data size cannot be greater than an aggregate SDU buffer size.

D-Series Systems Must Be Named

All D-series systems, unlike C-series systems, must be named, even if they are not part of an Expand network. Migrating your C-series application from an unnamed system to a named D-series system might cause problems if:

- ☐ Your application does not expect a file name to contain a system (node) name.
- ☐ The D-series system to which your C-series application is migrated is configured with 8-character (including the dollar sign, $) device names or 6-character (including the dollar sign, $) process names.
- ☐ Your program calls the DEFINEREADATTR procedure or the DEFINEINFO procedure.

File Names Always Include a System Name

If your C-series application does not expect file names returned to Guardian procedures to include a system (node) name, buffers allocated for file names might not be large enough to include the system (node) name.

For example, your C-series application might use the FNAMECOLLAPSE procedure to convert the internal format of a file name into its external format. If your application assumes that it is running on an unnamed system, it might allocate insufficient buffer space for the fully-qualified file name returned by the procedure.

To ensure that your C-series application runs successfully on a D-series system, modify your application to allow room for the system (node) name in buffers that might contain a file name returned from a Guardian procedure.

Device Names Should Not Exceed 7 Characters

Some applications will work incorrectly when the system is given a name, if the application provides for device names or process names that exceed 7 characters (including the dollar sign, $).

To ensure that your C-series application runs successfully on a D-series system, modify your application and your system configuration so that device names and process names do not exceed 7 characters, including the dollar sign ($).

DEFINEREADATTR and DEFINEINFO Return a System Name

The D-series DEFINEREADATTR and DEFINEINFO procedures always include the system (node) name in file names returned to an application. The =_DEFAULTS VOLUME attribute is unchanged.

C-series DEFINEREADATTR and DEFINEINFO procedures return the system (node) name only for remote systems.

To ensure that your C-series application runs successfully on a D-series system, modify your application to allow room for the system (node) name in buffers that might contain a file name returned from a DEFINEREADATTR or DEFINEINFO procedure.

Temporary File Names Have 7 Digits

On D-series systems, the number of concurrent temporary files allowed per volume has been increased. As a consequence, temporary file names now have 7 digits in their name, after the pound sign (#).

On C-series systems, temporary file names had only 4 digits in their name

To ensure that your C-series application runs successfully on a D-series system, modify your application if:

- ☐ It expects the length of a temporary file name to be 4 digits.
- ☐ It expects the (first) four digits of a temporary file name to be unique.

System-Message Protocol for Process Pairs Includes CPU Down Message

On D-series systems, when a backup process takes over as the primary process (following return from the CHECKMONITOR procedure) after the CPU of the primary process fails, the backup process subsequently receives a CPU down (-2) system message if it is reading system messages. Sending a CPU down (-2) system message to the backup process ensures that the backup process sees all status messages not seen by the primary process.

On C-series systems, the CPU down (-2) system message is not sent to a backup process when it takes over as the primary process.

To ensure that your C-series application runs successfully on a D-series system, if it accepts system messages but does not account for the CPU down (-2) message, you might need to modify your application to take note of or react to the CPU down (-2) system message.

Pool Space Address Adjustment

Starting with the D20 release, the DEFINEPOOL and RESIZEPOOL procedures sometimes adjust the starting address of a pool for address alignment. These procedures also ensure that pool space overhead and adjustments for alignment do not cause the pool to extend beyond the address that is the sum of the address specified for the beginning of the pool plus the specified pool size.

If your application attempts to calculate precisely the amount of pool space that it needs and does not allow for adjustment and alignment, it might not allocate adequate pool space.

To ensure that your C-series application runs successfully on a D-series system, it should not attempt to calculate precisely the amount of pool space that it needs (when using the DEFINEPOOL and RESIZEPOOL procedures).

TERMPROCESS Replaced by ATP6100

TERMPROCESS, which provided I/O support for asynchronous terminals, is not supported on D-series systems. On D-series systems, 6100/3600-class controllers are now the only option for asynchronous terminal support, and these controllers require ATP6100 rather than TERMPROCESS.

If your application performs I/O processing involving asynchronous terminals, you therefore might need to:

- ☐ Change your program to accommodate the longer device names supported by ATP6100.
- ☐ Be aware of the differences between TERMPROCESS and other protocols.

Device and Subdevice Names for ATP6100

TERMPROCESS supports terminal names consisting of a maximum of 8 characters, including the dollar sign ($), with the following format:

```
$device
```

ATP6100 supports terminal names with the following format:

```
$device.#subdevice
```

where the device name consists of a maximum of 8 characters including the dollar sign ($) and the subdevice consist of a maximum of 8 characters, including the pound sign(#).

Applications that support the TERMPROCESS format but do not support the ATP6100 format may have insufficient buffer space declared for terminal names.

To ensure that your C-series application runs successfully on a D-series system do one of the following:

- ☐ Have your system configured to keep $*device*.#*subdevice* to 8 or fewer characters.
- ☐ Modify your application to support the ATP6100 terminal-name format.

Protocol Differences

Terminals supported by TERMPROCESS might function differently than they do when they are supported by ATP6100 and other protocols. For example, if you press a function key, TERMPROCESS does not echo a line feed (LF) code, whereas some other protocols do:

TACL dialogue when TERMPROCESS is used:

```
5>  #PUSH F6
6>  <press the F6 key>
7>
```

TACL dialogue when a different protocol is used:

```
5>  #PUSH F6
6>  <press the F6 key>

7>
```

Nowait Write Buffer Integrity

A program using nowait write operations with Guardian procedures might incorrectly allow the contents of the buffer being written to be altered before completion of the operation (with an AWAITIO[X] procedure call).

For both C-series systems and D-series systems, it is possible for data to be moved from a write buffer after a write procedure call finishes executing but before the AWAITIO procedure finishes executing (thereby completing the operation). If an application alters the contents of such a buffer before the data is moved, the altered data might be moved instead of the original data (or the data that is moved might be a combination of the original data and the altered data).

It is more important for D-series systems than for C-series systems to be certain that buffers containing data to be written remain unaltered until the write operation is completed by the AWAITIO[X] procedure.

Note SETMODE 72 does not solve this problem. SETMODE 72 controls the use of buffers for read operations but not for write operations.

To ensure that your C-series application runs successfully on a D-series system, modify your application so that buffers containing data to be written remain unaltered until the AWAITIO procedure completes the write operation.

TMF Transactions Not Propagated to Device Simulator Process Automatically

In D-series systems, unlike C-series systems, device simulator processes (processes with device subtype 30) no longer have TMF transactions propagated to them as the default interprocess-message transfer mode. This change makes device simulator more correct because real devices do not have TMF transactions propagated to them.

Note This change also applies to spooler collectors (subtype 31) and tape simulators, although these two kinds of processes are not usually user-written.

To ensure that your C-series application runs successfully on a D-series system if it requires TMF transactions to be propagated to a device simulation process (subtype 30), modify your application to use SETMODE 117 with parameter 1 equal to 0.

Enhanced Attribute Values Returned from DEFINEs

Some of the attribute values returned by the DEFINE support procedures DEFINEREADATTR and DEFINEINFO on C-series systems have been enhanced for D-series systems.

Attributes DEFINEREADATTR and DEFINEINFO	Attribute values returned on C-series system	Attribute values returned on D-series system
File and subvolume names	All attributes representing file names or subvolume names on the local system, do not include the system (node) name (regardless of the current default system).	All attributes representing file names or subvolume names are in fully qualified external format including the system (node) name (except the VOLUME attribute of the CLASS DEFAULTS DEFINE).
Zero suppression	Some numeric attributes are not returned as zero suppressed.	All numerical attributes are returned as left-justified, zero-suppressed, variable length strings.

To ensure that your C-series application runs successfully on a D-series system if it depends on these attribute characteristics, modify your application to accommodate the new characteristics.

For a Process ID of 255 it is Important to Know the Source System

PIN 255 is reserved on D-series systems as the PIN part of a synthetic process ID. Synthetic process IDs are returned for high-PIN processes by some procedures and system messages that return only C-series process IDs. However, PIN 255 is still a valid PIN on C-series systems. Your application might need to process a process ID with a PIN of 255 in a different way depending on whether the process it identifies is a D-series process running in a high PIN or a C-series process.

You can determine the version of the operating system as follows:

1. Determine the node number of the system on which the process executes. In all practical situations you will have access to a C-series process ID in the remote form. The node number is in word 0, bits <8:15>.
2. Use the REMOTETOSVERSION procedure to determine if the remote system is running a C-series or D-series version of the operating system.

Glossary

accelerate. To use the Accelerator program to generate an accelerated object file.

accelerated object code. The RISC instructions that result from processing a TNS object file with the Accelerator.

accelerated object file. The object file that results from processing a TNS object file with the Accelerator. An accelerated object file contains the original TNS object code, the accelerated object code and related address map tables, and any Binder and symbol information from the original TNS object file.

Accelerator. A program that processes a TNS object file and produces an accelerated object file. Most TNS object code that has been accelerated runs faster on TNS/R processors than TNS object code that has not been accelerated.

ancestor. The process that is notified when a named process or process pair is deleted. It is the process that created the named process or process pair.

application. One or more processes that achieve a specific objective. Multiple processes in an application often communicate with each other using the message system and file system. See also "program" and "process."

C-series-compatible interface. The set of procedure calls, system messages, and event-message tokens available on a D-series system that permits unconverted applications to execute.

C-series process file name. A 12-word internal-format file name that identifies a process either by name or by CPU, PIN on a C-series system or on a D-series system using the C-series-compatible interface.

C-series system. A system that is running a C-series version of the operating system.

CISC. See "complex instruction-set computing (CISC)."

complex instruction-set computing (CISC). A processor architecture based on a large instruction set, characterized by numerous addressing modes, multicycle machine instructions, and many special-purpose instructions. Contrast with "reduced instruction-set computing (RISC)."

converted application. In the context of operating system releases, an application that has been modified to use extended features of the D-series operating system (for example, to run at a high PIN).

CPU, PIN. A C-series process identifier that is an 8-bit CPU number and an 8-bit process number. It is sometimes called a PID.

creation process ID (CRTPID). See "process ID."

creator. The process that initiates the execution of another process. Compare with "mom" and "ancestor."

CRTPID. See "process ID."

D-series enhanced interface. The set of procedure calls, system messages, and event-message tokens available on a D-series system that enable an application to take

advantage of the extended system limits. Applications must be converted to make use of this interface.

D-series process file name. A variable-length string that identifies a process by name or CPU, PIN and can also include the node name, sequence number, and one or two qualifiers.

D-series system. A system that is running a D-series version of the operating system.

device. An addressable I/O device, independent of its physical environment (for example, a terminal or printer).

Distributed Systems Management (DSM). A set of software tools that are used in the management of systems and networks. These tools include the ViewPoint console application, the Subsystem Control Facility (SCF), the Subsystem Programmatic Interface (SPI), the Event Management Service (EMS), the Distributed Name Service (DNS), and the token-oriented programmatic interfaces to the management processes for various Tandem subsystems.

Distributed Systems Management (DSM) application. An application that issues DSM tools to issue programmatic commands to a subsystem or retrieve event messages from a subsystem (or both) to assist in managing a system or network. In the requester-server approach, a management application is the requester and a subsystem is the server.

DSM. See "Distributed Systems Management (DSM)."

EMS. See "Event Management Service (EMS)."

Event Management Service (EMS). A set of software tools that provide event-message collection, logging, and distribution for the operating system. EMS provides different descriptions of events for operators and management applications and allows an operator or application to select specific event messages using EMS filters. EMS has programmatic interfaces based on SPI for both event reporting and event retrieval.

event message. In Distributed Systems Management (DSM), a special type of Subsystem Programmatic Interface (SPI) message that describes an event occurring in the system or network.

extended data segment. One or more consecutive absolute segments that are dynamically allocated by a process.

FCB. See "file control block (FCB)."

file. An entity that can be a disk file (all or part of a disk volume), an I/O device (such as a printer or terminal), or a process (an executing program).

file control block (FCB). (1) A data structure automatically created and managed by the file system that contains a collection of information about an open file. (2) A data structure on the user's data stack used by sequential I/O (SIO) to access SIO files. These FCBs contain information in addition to the information kept in the FCB automatically created and managed by the file system.

file number. An integer that represents a particular instance of an open of a file. A file number is returned by an open procedure and is used in subsequent I/O procedures to access the file.

file system. A set of operating system procedures and data structures that allows communication between a process and a file (disk file, I/O device, or another process).

filter. In the Event Management Service (EMS), a file that contains a list of criteria against which incoming event messages can be compared. Messages are either passed or not passed based on the list of criteria.

GMOM. See "godmother (GMOM)."

godmother (GMOM). A process that is notified when a process that is part of a job is deleted. The godmother of a process is the process that created the job to which the process belongs.

GPLDEFS file. A source file provided by Tandem that contains LITERAL and DEFINE declarations and data structures that are available for applications to use with sequential I/O (SIO) procedures.

high PIN. A process identification number (PIN) that is greater than 255. Contrast with "low PIN."

inherited force-low characteristic. A characteristic of a process that forces its child processes into low PINs when set. This characteristic is inherited from the creator of the process so that low-PIN processes can always communicate with their descendents. The characteristic can be overridden.

input/output process (IOP). A system process that controls one or more I/O units attached to the central processing unit (CPU) through I/O channels.

interprocess communication. The exchange of messages between processes in a system or network.

IOP. See "input/output process (IOP)."

low PIN. A process identification number (PIN) that ranges from 0 through 254. Contrast with "high PIN."

management application. See "Distributed Systems Management (DSM) application."

message. See "system message" and "SPI message."

message system. A set of operating system procedures and data structures that handle the mechanics of exchanging system messages between processes.

millicode. RISC instructions that implement various TNS low-level functions such as exception handling, real-time translation routines, and library routines that implement the TNS instruction set. Millicode is functionally equivalent to TNS microcode.

mom. A process that is notified when certain other processes are deleted. If a process is part of a process pair, the mom of the process is the other member of the pair. When a process is unnamed, its mom is usually the process that created it.

named process. A process to which a name was assigned when the process was created. Contrast with "unnamed process."

nested error list. In Distributed Systems management (DSM), an error list within another error list. When an error in one subsystem or in a library procedure prevents

another subsystem from performing a command, the calling subsystem reports this error by nesting error lists in its own response.

node. A system of one or more processor modules. Typically, a node is linked with other nodes to form a network.

node name. The portion of the file name that identifies the system through which the file can be accessed.

object file. A file generated by a compiler or Binder that contains machine instructions and other information needed to construct the executable code spaces and initial data for a process. The file may be a complete program that is ready for immediate execution, or it may be incomplete and require binding with other object files before execution.

object-file attributes. Flags in an object file that specify characteristics about the file or about its running as a process.

PFS. See "process file segment (PFS)."

physical memory. The semiconductor memory that is part of every processor module.

PID. See "CPU, PIN."

PIN. See "process identification number (PIN)."

process. An instance of execution of a program.

process descriptor. A process identifier returned by a system procedure call. It always contains the node name and sequence number as well as the process name or CPU, PIN designation. Contrast with "D-series process file name" and "C-series process file name."

process file name. See "D-series process file name" and "C-series process file name."

process file segment (PFS). An extended data segment that is automatically allocated to every process and contains operating system data structures, file-system data structures, and memory-management data structures.

process handle. A D-series 20-byte data structure that identifies a named or unnamed process in the network. A process handle identifies an individual process; thus, each process of a process pair has a unique process handle.

process ID. A C-series 4-word process identifier. A process ID contains a central processing unit (CPU) number, process identification number (PIN), creation timestamp or process name, and system number (optional). It is sometimes called a creation timestamp process ID (CRTPID).

process identification number (PIN). An unsigned integer that identifies a process in a processor module. Internally, a PIN is used as an index to the process control block (PCB) table.

process name. A name that is assigned to a process when the process is created. A process name uniquely identifies a process (or process pair) in a system.

process string. A process identifier that is suitable to display or print. It contains either the process name or the CPU, PIN, optionally preceded by the node name.

program. A static set of instruction codes and initialized data (for example, the output of a compiler or the Binder program) that is not executing. A program usually resides in a program file on disk. See also "process."

program file. An executable object file. See "object file."

reduced instruction-set computing (RISC). A processor architecture based on a relatively small and simple instruction set, a large number of general-purpose registers, and an optimized instruction pipeline that supports high-performance instruction execution. Contrast with "complex instruction-set computing (CISC)."

request. A message formatted and sent to a server by a requester. Requests also include status messages, such as CPU up and CPU down messages, that are placed on the process message queue of a process by the operating system. Contrast with "response."

requester. The process that initiates interprocess communication by sending a message to another process (usually a server). Contrast with "server."

response. A message formatted and sent to a requester by a server, usually to answer a request. In Distributed Systems Management (DSM), a response is the information or confirmation supplied by a subsystem in reaction to a command. A response is typically sent as one or more interprocess messages from a subsystem to a management application. Contrast with "request."

RISC. See **"**reduced instruction-set computing (RISC)."

RISC instructions. Register-oriented 32-bit machine instructions that are directly executed on TNS/R processors. RISC instructions execute only on TNS/R systems, not on TNS systems. Contrast with "TNS instructions."

segment. A unit of storage in memory.

segment ID. An integer that a process uses to identify an extended data segment. It can also specify the type of extended data segment for a C-series system.

sequential I/O (SIO) procedures. A set of related operating system procedures that are used for reading and writing sequential files.

server. The process that receives, acts upon, and replies to messages from requesters. Contrast with "requester."

simple token. In the System Programmatic Interface (SPI), a token consisting of a token code and a value of the type indicated in the token code. Although simple token values can have an internal structure, SPI stores and retrieves those values without any knowledge of their structure.

SIO procedures. See "sequential I/O (SIO) procedures."

SPI. See "Subsystem Programmatic Interface (SPI)."

SPI message. A message specially formatted by the Subsystem Programmatic Interface (SPI) procedures for communication between a management application and a subsystem, or between one subsystem and another. An SPI message consists of a collection of tokens.

structured token. In the Subsystem Programmatic Interface (SPI), a token whose value is a structure. Some structured tokens are simple tokens with fixed structures, while other structured tokens are extensible and can be extended by adding new fields at the end.

subsystem. A program or a set of processes or procedures that manages a cohesive set of objects (for example, a set of files or devices). Each subsystem has a process from which applications can request services by sending commands (in some cases, this process is the entire subsystem). See "Distributed Systems Management (DSM) application."

Subsystem Programmatic Interface (SPI). A common, message-based interface that can be used to build and decode messages used for communication between requesters (for example, a management application) and servers (Tandem subsystems). SPI includes procedures to build and decode specially formatted messages; source definition files in TAL, COBOL85, Pascal, C, and TACL; and definition files in DDL for programmers writing their own subsystems.

swap files. The disk files to and from which data is copied during swapping, which is the process of copying data between physical memory and disk storage.

synthetic process ID. A process name or timestamp followed by the central processing unit (CPU) number and a process identification number (PIN) value of 255. The use of a synthetic process ID is limited; it allows an unconverted process on a D-series system to support high-PIN openers.

system message. A block of information, usually in the form of a structure, that is sent from one process to another process.

system process. A process whose primary purpose is to manage system resources such as memory or I/O devices. It is essential to a system-provided service, and failure of a system process can cause the processor module to fail. Most system processes are automatically created when the processor module is cold loaded. Contrast with "user process."

Tandem NonStop Series (TNS). Tandem computers that support the operating system and that are based on complex instruction-set computing (CISC) technology. TNS processors implement the TNS instruction set. Systems with these processors are the NonStop II, NonStop TXP, NonStop EXT, NonStop VLX, NonStop Cyclone, NonStop CLX 600, CLX 700, and CLX 800, and NonStop CLX/R 1100 systems. Contrast with "Tandem NonStop Series/RISC (TNS/R)."

Tandem NonStop Series/RISC (TNS/R). Tandem computers that support the operating system and that are based on reduced instruction-set computing (RISC) technology. TNS/R processors implement the RISC instruction set and are upwardly compatible with the TNS system-level architecture. Systems with these processors are the

NonStop Cyclone/R, NonStop CLX 2000, and NonStop CLX/R 1200 systems. Contrast with "Tandem NonStop Series (TNS)."

TNS. See "Tandem NonStop Series (TNS)."

TNS instructions. Stack-oriented, 16-bit machine instructions defined as part of the TNS environment. On TNS systems, TNS instructions are implemented by microcode; on TNS/R systems, TNS instructions are implemented by millicode routines or by translation to an equivalent sequence of RISC instructions. Contrast with "RISC instructions."

TNS object file. The object file created by a TNS compiler. The file contains TNS instructions and other information needed to construct the code spaces and the initial data for a TNS process.

TNS/R. See "Tandem NonStop Series/RISC (TNS/R)."

token. In SPI, a distinguishable unit of data in an SPI message. A token has two parts: an identifying code (a token code or token map) and a token value.

unconverted application. In the context of operating system releases, an application that has not been modified to use extended features of the D-series operating system.

unnamed process. A process to which a name was not assigned when the process was created. Contrast with "named process."

user process. A process whose primary purpose is to solve a user's immediate problem. A user process is not essential to the availability of a processor module. A user process is created only when the user explicitly creates it. Contrast with "system process."

ZSYSDDL file. A file provided by Tandem that contains DDL definitions of source declarations for Guardian system procedures and system messages. ZSYSDDL is used to generate the ZSYSTAL, ZSYSCOB, ZSYSC, and ZSYSPAS files for use with TAL, COBOL85, C, and Pascal applications, respectively.

$RECEIVE. A special file name through which a process receives and optionally replies to messages from other processes.

Index

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Special characters