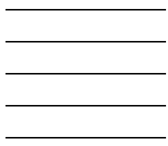


NonStop TS/MP Pathsend and Server Programming Manual

Abstract

This manual describes how to write two types of programs as part of a Pathway application: requester programs that use the Pathsend application program interface (API) and server programs that service requests from all types of Pathway requesters.

Product Version

NonStop TS/MP D44

Supported Releases

This manual supports D44.00 and all subsequent D4x releases and G03.00 and all subsequent G-series releases until otherwise indicated in a new edition.

Part Number	Published	Release ID
132500	July 1997	D44.00

Document History

Part Number	Product Version	Published
110074	NonStop TS/MP D31	July 1995
123813	NonStop TS/MP D31	December 1995
132500	NonStop TS/MP D44	July 1997

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).

Ordering Information

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

Document Disclaimer

Information contained in a manual is subject to change without notice. Please check with your authorized Tandem representative to make sure you have the most recent information.

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.

U.S. Government Customers

FOR U.S. GOVERNMENT CUSTOMERS REGARDING THIS DOCUMENTATION AND THE ASSOCIATED SOFTWARE:

These notices shall be marked on any reproduction of this data, in whole or in part.

NOTICE: Notwithstanding any other lease or license that may pertain to, or accompany the delivery of, this computer software, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Section 52.227-19 of the FARS Computer Software—Restricted Rights clause.

RESTRICTED RIGHTS NOTICE: Use, duplication, or disclosure by the Government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013.

RESTRICTED RIGHTS LEGEND: Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraphþ(b)(3)(B) of the rights in Technical Data and Computer Software clause in DAR 7-104.9(a). This computer software is submitted with “restricted rights.” Use, duplication or disclosure is subject to the restrictions as set forth in NASA FARþSUP 18-52þ227-79 (Aprilþ1985) “Commercial Computer Software—Restricted Rights (Aprilþ1985).” If the contract contains the Clause at 18-52þ227-74 “Rights in Data General” then the “Alternate III” clause applies.

U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract.

Unpublished — All rights reserved under the Copyright Laws of the United States.

New and Changed Information

This is the third edition of the *NonStop TS/MP Pathsend and Server Programming Manual*.

Writers of Pathsend requesters and all Pathway servers should read this manual. Writers of SCREEN COBOL requesters should read the *Pathway/TS TCP and Terminal Programming Guide*.

This third edition includes changes to reflect product changes, and also additional enhancements. Substantive changes (changes that are not simply editorial) are marked by change bars in the right-hand margin of the page.

Product Changes

This manual documents the following changes to NonStop TS/MP and related products:

- New Pathsend procedure calls have been added to support context-sensitive communication (dialogs) between requesters and servers.

  Section 3, Writing Pathsend Requesters, now describes how to use these procedure calls in a Pathsend requester. Section 5, Pathsend Procedure Call Reference, provides detailed syntax for the procedure calls. Section 6, Pathsend Errors, describes new error messages related to the calls.

- A new TUXEDO to Pathway translation server is now available as part of Release 2 of the NonStop TUXEDO system. This translation server allows TUXEDO requesters (that is, TUXEDO clients and TUXEDO servers acting as clients) to access Pathway servers.

  Other Transaction Processing Environments on page 1-13 and Writing Pathway Servers That Interoperate With TUXEDO Requesters on page 4-17 now mention the availability of the TUXEDO to Pathway translation server. Details on the use of this translation server are provided in the *NonStop TUXEDO System Pathway Translation Servers Manual*.

- Changes are being made to the Remote Server Call (RSC) product, and some of the protocols formerly listed in Section 2 of this manual may no longer be supported. Therefore, all mention of support for specific platforms and protocols under Clients Using RSC and POET on page 2-15 has been removed and replaced by a more general statement. For detailed information about the platforms and protocols supported by the RSC product, refer to the *Remote Server Call (RSC) Programming Manual*.

Enhancements to the Manual

The following enhancements have been made to the material in this manual:

- A new subsection, Consideration for Servers Used With Remote Server Call (RSC) Clients on page 4-4, has been added to mention the optional server reply code for servers used with Remote Server Call (RSC) clients. For details, refer to the *Remote Server Call (RSC) Programming Manual*.
- Corrections have been made to the discussion of Server-Class Send Timeout on page 5-27.
- Additional cause information has been added to the description of Pathsend error 902 on page 6-3.
- Several minor technical corrections and editorial changes have been made.

Contents

New and Changed Information iii

About This Manual xi

Notation Conventions xvii

1. Introduction to Pathway Application Programming

Which Sections Do You Need? 1-1
Advantages of the Pathway Environment 1-3
- Ease of Development 1-3
- Manageability 1-4
- Data Integrity 1-4
- Fault Tolerance 1-5
- Other Tandem Fundamentals 1-6

Pathway Applications 1-7
- Servers and Server Classes 1-8
- Requesters 1-9

The Pathsend Environment 1-10
- Pathsend Processes 1-10
- LINKMON Processes 1-11

Client/Server Capabilities 1-12
Other Transaction Processing Environments 1-13
Development Tools and Utilities 1-14
- Programming Languages and Related Tools 1-14
- The Inspect Symbolic Debugger 1-14
- The Pathmaker Application Generator 1-14
- Client/Server Development Tools 1-15

Transaction Processing Scenario 1-15

2. Designing Your Application

Designing Transactions 2-1
- Analyzing Data Flow 2-2
- Identifying Transaction Components 2-4
- Protecting Transactions 2-6

Designing the Database 2-9
- Logical Design 2-9
- Physical Design 2-10
- Database Managers 2-10
- Remote Duplicate Database Facility (RDF) 2-11

2. Designing Your Application (continued)

Designing Requester Programs 2-11
SCREEN COBOL Requesters 2-12
IDS Requesters 2-12
Pathsend Requesters 2-13
Clients Using RSC and POET 2-15
Requesters Using GDSX 2-16
Dividing Function Between Requester and Server 2-19
Designing Server Programs 2-19
Design Considerations 2-20
Server Program Structure 2-25
Designing Applications for Batch Processing 2-27

3. Writing Pathsend Requesters

The Pathsend Procedure Calls 3-1
Interprocess Communication in the Pathsend Environment 3-2
Basic Pathsend Programming 3-3
Programming for Failure Recovery 3-3
Security Issues 3-6
Avoiding Coded PATHMON Names 3-7
Context-Sensitive Pathsend Programming 3-8
Using Context-Sensitive Requesters With Context-Free Servers 3-8
Resource Utilization 3-8
Programming for Failure Recovery 3-9
Cancellation of Server-Class Send Operations 3-10
Writing Requesters That Interoperate With NonStop TUXEDO Servers 3-11

4. Writing Pathway Servers

Basic Pathway Server Programming 4-1
Servers Shared by Different Types of Requesters 4-1
Guardian Servers and Pathway Servers 4-2
Server Stop Protocol 4-2
Handling of Messages from $RECEIVE 4-2
Pathsend Requester Failures 4-2
LINKMON Process Failures 4-3
Linkage Space Considerations 4-3
Considerations for Servers Used With SCREEN COBOL Requesters 4-3
Consideration for Servers Used With Remote Server Call (RSC) Clients 4-4
Nested Servers 4-4
Using Context-Free Servers With Context-Sensitive Requesters 4-4

4. Writing Pathway Servers (continued)

Considerations for Servers That Use the TMF Subsystem 4-5
- Recommended Structure for Applications 4-5
- Writing a Server to Use the TMF Subsystem 4-6
- Using Audited and Nonaudited Files 4-7
- Locking Records 4-8
- Grouping Transaction Operations 4-8
- Servers as Process Pairs 4-10
- Transaction Deadlocks 4-10

Considerations for Debugging Pathway Servers 4-11
- LINKMON Process and TCP Timeouts 4-11
- PATHMON Process Timeouts 4-12
- Server Timeouts 4-12
- Avoiding Timeout Errors 4-12

Writing Context-Sensitive Servers 4-13
- Functions of a Context-Sensitive Server 4-13
- Detecting a Newly Established Dialog 4-14
- Receiving, Servicing, and Replying to Messages in a Dialog 4-14
- Correlating Messages With a Dialog 4-16
- Continuing a Dialog 4-16
- Aborting a Dialog 4-16
- Terminating a Dialog 4-16
- Detecting an Aborted Dialog 4-16

Writing Pathway Servers That Interoperate With TUXEDO Requesters 4-17

5. Pathsend Procedure Call Reference

Calls From C or C++ 5-2

Calls From COBOL85 5-3

Calls From Pascal 5-4

Calls From TAL or pTAL 5-5

SERVERCLASS_DIALOG_ABORT_ Procedure 5-6
- Syntax 5-6
- Considerations 5-6

SERVERCLASS_DIALOG_BEGIN_ Procedure 5-7
- Syntax 5-7
- Considerations 5-11

SERVERCLASS_DIALOG_END_ Procedure 5-12
- Syntax 5-12
- Considerations 5-12

5. Pathsend Procedure Call Reference (continued)

SERVERCLASS_DIALOG_SEND_ Procedure 5-13
Syntax 5-13
Considerations 5-16
SERVERCLASS_SEND_ Procedure 5-17
Syntax 5-17
Considerations 5-20
SERVERCLASS_SEND_INFO_ Procedure 5-21
Syntax 5-21
Considerations 5-22
Usage Considerations for Pathsend Procedures 5-23
Condition Code 5-23
Waited I/O 5-23
Nowait I/O 5-23
Calls Within a TMF Transaction 5-24
Server-Class Send Operation Number 5-24
Timeout Considerations for Pathsend Programming 5-27

6. Pathsend Errors

Types of Errors Returned by the Pathsend Procedures 6-1
Descriptions of Pathsend Errors 6-1

A. NonStop TS/MP Limits for Pathsend Requesters

B. Examples

Pathsend Requester Example B-1
Nested Server Example B-53

Glossary

Index

Examples

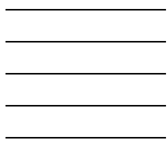
Example 2-1. Sample Pathsend Requester Program Structure 2-15
Example 2-2. COBOL85 Server Program Example 2-25
Example B-1. Context-Free Pathsend Requester Program B-2
Example B-2. Context-Free Server Program B-53

Figures

Figure i. Related Documentation xiii
Figure 1-1. Pathsend Interprocess Communication 1-11
Figure 1-2. Example Application Using a Pathsend Requester 1-16
Figure 2-1. Data Flow for a Business Task 2-3
Figure 2-2. Relationships Between Transaction Functions 2-5
Figure 2-3. Pathway Application Programming for the TMF Subsystem 2-7
Figure 2-4. GDSX as a Front-End Process 2-18
Figure 2-5. GDSX as a Back-End Process 2-24

Tables

Table 1-1. Task and Manual Correspondences 1-2
Table 2-1. Considerations for Requester Programs 2-11
Table 4-1. Meaning of Error Codes Returned by Context-Sensitive Server in Reply 4-15
Table 5-1. Summary of Pathsend Procedure Calls 5-1
Table A-1. Limits for Pathsend Requesters A-1



About This Manual

This section describes the purpose and the contents of this manual and of other manuals closely related to this manual.

This manual is one of a set of manuals that describe the NonStop Transaction Services/MP (NonStop TS/MP) and Pathway/Transaction Services (Pathway/TS) products. The contents of these products are as follows:

- NonStop TS/MP: This product consists of the PATHMON process, the LINKMON process, the PATHCOM process and interface, and the Pathsend procedure calls.
- Pathway/TS: This product consists of the terminal control process (TCP), the SCREEN COBOL compiler and run-time environment, and the SCREEN COBOL Utility Program (SCUP).

Together with the NonStop Transaction Manager/MP (NonStop TM/MP) product, the NonStop TS/MP product forms the foundation for Tandem's open transaction processing and client/server products on NonStop Himalaya systems. Such products include the NonStop TUXEDO system, the Remote Server Call (RSC) product, and the Pathway Open Environment Toolkit (POET).

The Pathway/TS product supports requester programs that run in the Guardian environment and communicate with terminals and intelligent devices. It requires the services of the NonStop TS/MP product.

Purpose of This Manual

This manual is a combined reference manual and programming guide. It contains reference information about the Pathsend procedure calls, and it also contains information about how to design and code applications using these calls.

In addition, this manual contains information about how to design and code Pathway servers to be used with all types of requesters and clients.

Who Should Read This Manual

This manual is intended for programmers writing Pathway applications that include Pathsend requesters, Pathway servers, or both. Readers should be experienced programmers familiar with the Guardian environment and the Tandem programming languages they are using.

What Is in This Manual

This manual contains application design and programming information for writers of Pathsend requesters and Pathway servers. This information includes syntax, usage, and error-handling information for the Pathsend procedure calls. The manual has the following structure:

- Section 1, Introduction to Pathway Application Programming, contains an overview of Pathway application programming, with emphasis on Pathsend requesters and Pathway servers.
- Section 2, Designing Your Application, provides information about Pathway application design, with emphasis on Pathsend requesters and Pathway servers.
- Section 3, Writing Pathsend Requesters, describes how to write requesters that use the Pathsend procedure calls.
- Section 4, Writing Pathway Servers, describes how to write Pathway servers.
- Section 5, Pathsend Procedure Call Reference, provides the syntax, parameter descriptions, and coding considerations for the Pathsend procedure calls.
- Section 6, Pathsend Errors, provides cause, effect, and recovery information for all Pathsend error codes.
- Appendix A, NonStop TS/MP Limits for Pathsend Requesters, summarizes the limits that apply to the Pathsend programming environment.
- Appendix B, Examples, provides source code for an example Pathsend requester and an example Pathway server.
- A standard Glossary for all NonStop TS/MP and Pathway/TS manuals defines all terms relevant to these products.

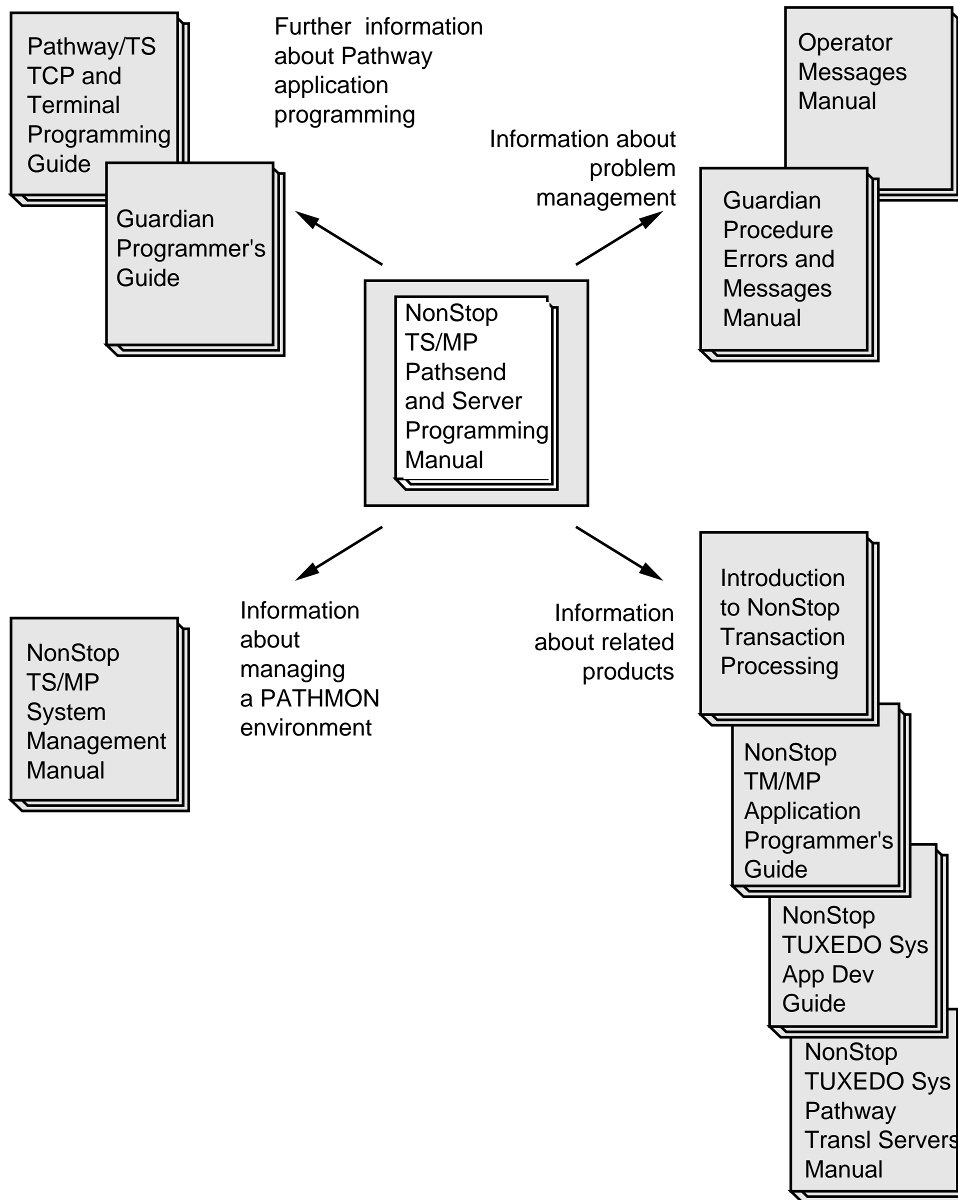
Related Documentation

This manual is one in a set of Tandem manuals for the NonStop TS/MP and Pathway/TS products. Figure i, Related Documentation, shows the manuals that are most closely related to this manual.

The following paragraphs describe each of the supporting manuals shown in Figure 1.

- *Pathway/TS TCP and Terminal Programming Guide*. This guide describes how to write requester programs using SCREEN COBOL.
- *Guardian Programmer's Guide*. This guide provides information about programming in the Guardian environment, including use of the Guardian procedure calls. It is useful to programmers who are writing Pathway servers, especially if they are writing them in C, C++, the Transaction Application Language (TAL), or the Portable Transaction Application Language (pTAL).

Figure i. Related Documentation



For information about informational, warning, and error messages, refer to the following manuals:

- *Guardian Procedure Errors and Messages Manual*. This manual describes the Guardian messages for Tandem systems that use the Tandem NonStop Kernel. The manual covers various types of error codes and error lists associated with Guardian procedure calls and also the interprocess messages sent to application programs by the operating system and the command interpreter.

- *Operator Messages Manual*. This manual describes system messages. For each message the manual provides an explanation of the cause, a discussion of the effect on the system, and suggestions for corrective action. The "PATHWAY Messages" section describes the operator messages generated by the PATHMON environment.

For information about managing the PATHMON environment in which your Pathway applications run, you might want to read the following manual:

- *NonStop TS/MP System Management Manual*. This manual describes how to start, configure, and manage a PATHMON environment. This manual also includes information about monitoring and adjusting your PATHMON environment to optimize performance, suggestions for diagnosing and fixing problems, and manageability guidelines such as how to start objects in parallel to improve performance. It also describes the commands for configuring, starting, and managing a PATHMON environment.

For information about related Tandem products, see the following publications:

- *Introduction to NonStop Transaction Processing*. This manual describes the architecture, components, and benefits of Tandem transaction processing products, including NonStop TS/MP, Pathway/TS, and related products such as the NonStop TUXEDO system.
- *NonStop TM/MP Application Programmer's Guide*. This guide provides information about programming for the Transaction Management Facility (TMF) subsystem, including use of the TMF procedure calls. It is useful to programmers who are writing Pathsend requesters and Pathway servers, especially if they are writing them in C, C++, the Transaction Application Language (TAL), or the Portable Transaction Application Language (pTAL).
- *NonStop TUXEDO System Application Development Guide*. This guide provides information about writing NonStop TUXEDO applications. Pathsend requesters can interoperate with NonStop TUXEDO applications either directly by using the NonStop TUXEDO Application Transaction Monitor Interface (ATMI) functions, or indirectly by using the Pathway translation server for the NonStop TUXEDO system.
- *NonStop TUXEDO System Pathway Translation Servers Manual*. This manual provides information about writing Pathsend and SCREEN COBOL requesters that interoperate with NonStop TUXEDO servers by using the Pathway to TUXEDO translation server, and information about writing Pathway servers that interoperate with TUXEDO requesters (clients or servers acting as clients) by using the TUXEDO to Pathway translation server. It also provides configuration information for the two translation servers.

Other Manuals in the Manual Set

In addition to the NonStop TS/MP and Pathway/TS manuals shown in Figure i, the manual set for these products includes the following manuals:

- *Pathway/TS System Management Manual*. This manual describes how to start, configure, and manage Pathway/TS objects (TCPs, terminal objects, SCREEN COBOL programs, and tell messages) in a PATHMON environment. It also describes the commands for configuring, starting, and managing Pathway/TS objects.
- *NonStop TS/MP Management Programming Manual*. This manual describes the management programming interface to the Pathway subsystem, including programmatic commands and related objects, event messages, and error lists.
- *Pathway/TS Management Programming Manual*. This manual describes the error lists and event messages issued by the management programming interface to the Pathway subsystem.

Other Manuals of Interest

Other manuals that might be of interest to readers of this manual include the following:

- *Availability Guide for Application Design*. This manual describes the features of Tandem NonStop systems that support the availability of applications. This manual includes a section about application availability in the Pathway transaction processing environment.
- *Guardian Programmer's Guide*. This guide provides information about programming in the Guardian environment, including use of the Guardian procedure calls. It is useful to programmers who are writing Pathway servers, especially if they are writing them in C, C++, the Transaction Application Language (TAL), or the Portable Transaction Application Language (pTAL).

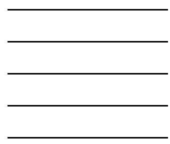
Your Comments Invited

After using this manual, please take a moment to send us your comments. You can do this by returning a Reader Comment Card or by sending an Internet mail message.

A Reader Comment Card is located at the back of printed manuals and as a separate file on the Tandem CD Read disc. You can either FAX or mail the card to us. The FAX number and mailing address are provided on the card.

Also provided on the Reader Comment Card is an Internet mail address. When you send an Internet mail message to us, we immediately acknowledge receipt of your message. A detailed response to your message is sent as soon as possible. Be sure to include your name, company name, address, and phone number in your message. If your comments are specific to a particular manual, also include the part number and title of the manual.

Many of the improvements you see in Tandem manuals are a result of suggestions from our customers. Please take this opportunity to help us improve future manuals.



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
```

Notation for Messages

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

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

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

p-register

process-name

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

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

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list might 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:

```
LDEV ldev [ CU %ccu | CU %... ] UP [ (cpu,chan,%ctlr,%unit) ]
```

{ } Braces. A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list might 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:

```
LBU { X | Y } POWER FAIL
```

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

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

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

% Percent Sign. A percent sign precedes a number that is not in decimal notation. The %þnotation precedes an octal number. The %Bþnotation precedes a binary number. The %Hþnotation precedes a hexadecimal number. For example:

```
%005400
```

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

Change Bar Notation

Change bars are used to indicate substantive differences between this edition of the manual and the preceding edition. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

> The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).
>
> The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

1 Introduction to Pathway Application Programming

This section introduces Pathway transaction processing applications, which you write and run with the assistance of the NonStop Transaction Services/MP (NonStop TS/MP) and Pathway/Transaction Services (Pathway/TS) software.

This section discusses the following topics:

- Advantages of the Pathway environment
- Pathway applications, including requesters and servers
- The Pathsend environment
- Client/server capabilities
- Other supported transaction processing environments
- Development tools and utilities
- A sample transaction processing scenario

This section does not describe the components of the NonStop TS/MP software in detail. You can find more detailed information about these components in the introductory sections of the *NonStop TS/MP System Management Manual*. The *Introduction to NonStop Transaction Processing* manual summarizes the main features and capabilities of the NonStop TS/MP software.

Which Sections Do You Need?

The remaining sections of this manual describe how to write two types of programs as part of a Pathway application: requester programs that use the Pathsend application program interface (API) and server programs that service requests from all types of Pathway requesters. The sections are organized into logical groups of information for easy reference. Depending on the types of requesters and servers in your Pathway application and which parts of the application you are working on, you might not need

to read all sections of this manual. Table 1-1 is a descriptive map listing which sections are relevant to particular programming tasks.

Table 1-1. Task and Manual Correspondences

If Your Application Includes…	You Need…	To Perform the Following…
Pathsend requesters	Section 2, Designing Your Application	Design an application including Pathsend requesters
	Section 3, Writing Pathsend Requesters	Write a Pathsend requester program
	Section 5, Pathsend Procedure Call Reference	Look up the syntax of Pathsend procedures
	Section 6, Pathsend Errors	Look up cause, effect, and recovery for errors returned to a Pathsend requester program
	Appendix A, NonStop TS/MP Limits for Pathsend Requesters	Look up limits pertaining to Pathsend requesters
	Appendix B, Examples	See examples of Pathsend requester programs
Pathway servers	Section 2, Designing Your Application	Design an application including Pathway servers
	Section 4, Writing Pathway Servers	Write a Pathway server program
	Appendix B, Examples	See examples of Pathway server programs

If you are writing SCREEN COBOL requesters, you need the *Pathway/TS TCP and Terminal Programming Guide* and the *Pathway/TS SCREEN COBOL Reference Manual* for programming information.

If you are writing Pathsend requesters that communicate with NonStop TUXEDO servers, or if you are writing Pathway servers that handle requests from NonStop TUXEDO requesters (clients or servers acting as clients), you also need the manuals for the NonStop TUXEDO system, particularly the *NonStop TUXEDO System Application Development Guide*, for additional information. If you are using the Pathway to TUXEDO translation server or the TUXEDO to Pathway translation server, you also need the *NonStop TUXEDO System Pathway Translation Servers Manual*. This manual provides configuration, startup, and programming information.

Advantages of the Pathway Environment

NonStop TS/MP provides ease of development, manageability, and the fundamental strengths and benefits of Tandem NonStop systems. The strengths and benefits of Tandem systems include data integrity, fault tolerance, high performance and low cost, system security, scalability, and distributed processing. The following paragraphs describe how NonStop TS/MP and related products—known together as the Pathway environment—benefit the application designer and programmer. The *Introduction to NonStop Transaction Processing* provides a fuller description of how all the Tandem fundamentals apply to transaction processing.

Ease of Development

Development costs are one of the highest expenses associated with online transaction processing (OLTP) systems. The more sophisticated the features and safeguards that are built into your OLTP application—for example, multiprocessing, fault tolerance, and data integrity—the greater the costs. When you use NonStop TS/MP and related Tandem transaction processing products to create your OLTP applications, development time and efforts, and therefore costs, can be measurably reduced.

This cost reduction occurs because:

- NonStop TS/MP and related products provide the most complex components of an OLTP application. NonStop TS/MP includes the transaction monitor (PATHMON), the command interpreter for management (PATHCOM), and the means for interprocess communication.

  In addition, the NonStop Transaction Manager/MP (NonStop TM/MP) product provides the transaction manager, and the Pathway/TS product provides a multithreaded terminal control process (TCP) for communication with terminals, including fault tolerance and transaction protection. (On Tandem NonStop system models earlier than the Himalaya systems, Pathway/TS is packaged as part of the Pathway transaction processing system.)

  Used with or without NonStop TM/MP and Pathway/TS, NonStop TS/MP provides a run-time Pathway environment to simplify your development efforts for scalable OLTP applications on a massively parallel processor architecture.

- Tandem makes valuable application development tools and utilities available for the Pathway environment. These development tools and utilities can significantly reduce the amount of programming time and effort required to generate a working Pathway application.

  The Remote Server Call (RSC) product facilitates client/server computing, allowing workstation applications to access Pathway servers. A large number of packaged tools and utilities are commercially available for use with RSC, including Tandem's Pathway Open Environment Toolkit (POET).

- The Pathway environment helps you standardize program code. You can repeat and reuse code; you do not have to write the same requester and server programs over and over again. This ability to reuse code saves development time.

- The Pathway environment allows you to isolate and test your requester and server programs before adding them to a running application. This capability is important because coding errors are difficult, time-consuming, and expensive to find after an application is put into production.
- OLTP products that are compatible with the Pathway environment are available from third-party vendors through the Tandem Alliance program.

In addition to making initial development faster and easier, the structured Pathway environment allows you to implement enhancements and develop new applications by simply adding new requesters, sharing existing servers, or adding new servers to the existing application. You can use code modules in the existing application as templates for new modules in the modified or new application.

Manageability

Online transaction processing operations present a dynamic environment in which hundreds of different transactions—from disparate locations and many different I/O devices—can be entered concurrently and processed within seconds. To process hundreds of transactions, thousands to millions more application program instructions must be executed. It is critical that you be able to control and monitor such a complex processing environment.

To control and monitor your Pathway environment—as well as simplify the task of system management—NonStop TS/MP provides the following:

- A PATHMON process, which provides a single point of control over your OLTP applications and operations
- A choice of two different system management interfaces: the interactive PATHCOM interface and the Subsystem Programmatic Interface (SPI)
- Status and error reporting capabilities, provided through a log file and through the Event Management Service (EMS)

Because NonStop TS/MP provides these processes and capabilities, you do not have to spend the time and money to develop, test, and implement comparable mechanisms.

For more information about the PATHMON process, the management interfaces, and status and error reporting capabilities in the Pathway environment, refer to the *NonStop TS/MP System Management Manual*, the *Pathway/TS System Management Manual*, the *NonStop TS/MP Management Programming Manual*, and the *Pathway/TS Management Programming Manual*.

Data Integrity

If your database is corrupted by a hardware or software failure, you might need weeks to isolate and then correct the problem. Because an inaccessible or inconsistent database can have a dramatic, adverse effect on business operations, Tandem developed the Transaction Management Facility (TMF) subsystem, provided in the NonStop TM/MP product, as a way of ensuring database consistency. The TMF subsystem, which works with NonStop TS/MP, protects the entire database from catastrophic system failures by maintaining an audit trail of database changes (that is, transactions); an audit trail is also

commonly known as a transaction log. You can use the audit trail to rebuild the database in the event of a hardware or software failure.

The design of Pathway servers supports the integrity of individual transactions and therefore transaction processing protection as a whole. Because the requester/server model allows a clear division of processing functions, application programmers can code each server program to handle a specific set of transaction types: for example, checking an account balance, entering a new customer, or updating the parts inventory. The server processes service their transactions by performing the same set of tasks over and over again. In this way, a valid transaction is defined as a specific set of tasks both by the requester program and within the server logic.

If for any reason a server is unable to complete all tasks involved in processing a transaction, it can abort the transaction and thereby maintain the transaction's integrity. The server does not have to wait for the requester to abort the transaction.

Fault Tolerance

Because OLTP systems automate core business operations and deliver key business services, companies depend on OLTP applications to stay up and running—even if a hardware or software component fails.

Tandem NonStop systems, which are specifically intended for online transaction processing, are designed to remain continuously available during the hours when transactions are being entered and business is being conducted. Typically, a Tandem NonStop system can continue processing despite the failure of any single software or hardware component within that system. This ability is referred to as fault tolerance.

In the Pathway environment, automatic fault tolerance (that is, fault tolerance that does not require any additional programming effort on your part) is provided by the use of process pairs and the actions of the PATHMON process, the TMF subsystem, and the terminal control process (TCP) provided with the Pathway/TS product.

In the Guardian operating environment, the functions and tasks of an application are performed by processes, which are running programs. A process pair consists of a primary process, which does some specific function in the overall work of the application, and a secondary (backup) process, which remains ready to take over if the primary process fails. During processing, the primary process keeps the backup process informed of what it is doing (for example, sending a request) by means of special interprocess messages, in an activity called checkpointing. Through checkpointing, the backup process has enough information to take over and continue if the primary process fails.

Both the PATHMON process and the TCP can be configured as process pairs to support Pathway applications. When the PATHMON process is configured as a process pair, you are ensured the ability to control and monitor OLTP system operation even if the primary PATHMON process fails. When a TCP is configured as a process pair and the primary TCP fails, terminals controlled by the TCP can still be used.

Pathway server classes provide additional fault tolerance by allowing requests to be rerouted to surviving server processes in a server class if one server process fails.

Besides process pairs and server classes, fault tolerance in a Pathway application is ensured by the PATHMON process, the TCP, and the TMF subsystem. Using information stored in the PATHMON configuration file, the PATHMON process automatically restarts processes at their initialization level after a failure, allowing these processes to resume work immediately.

Other Tandem Fundamentals

Besides data integrity and fault tolerance, the Pathway environment also provides the high performance and low cost, system security, scalability, and distributed processing of Tandem NonStop systems.

High Performance and Low Cost

The more transactions your system can process (preferably without degrading response time), the lower the cost of each transaction. The Pathway environment supports fast response time and high system throughput by allowing:

- Component processes in a Pathway application (for example, requester and server processes) to reside and execute concurrently in different processors of a multi-processor system or even a network. This is called multiprocessing.
- More than one Pathway application to run in a Tandem NonStop system.

NonStop TS/MP also supports fast response time and high system throughput by allowing the replication of processes and programs and the distribution of processes. For example:

- The PATHMON process can dynamically create additional copies of server processes at times of peak demand and delete the additional servers when activity slows again.
- You can add copies of requester and server programs to your Pathway application to maintain fast response time when the number of users or terminals increases.
- You can distribute processes such as requesters and servers close to the resources they manage, reducing interprocess communication time within a network.
- You can distribute requesters and servers to less active processors if peak activity on a particular processor is affecting throughput or response time.

System Security

The Guardian operating environment includes basic mechanisms for controlling access to files, whether they are data files or program files. Because NonStop TS/MP runs in the Guardian operating environment, Guardian system security parameters also apply to Pathway users and processes. In addition, you can supplement the security features of the Guardian environment with the Safeguard product, which provides authentication, authorization, and auditing capabilities for Guardian files.

Scalability

Your organization must be able to expand its transaction processing system as its operations evolve and its technical requirements change. Tandem NonStop systems are expressly designed to support incremental, modular expansion, allowing you to increase the size and processing power of your transaction processing system by:

- Adding hardware and application resources to your existing system
- Linking individual Pathway applications into a single network or adding more Pathway applications to an existing network
- Supporting an open systems architecture in which standards-based networks as well as devices and systems from other vendors can be connected to your Tandem system

Distributed Processing

Data communications technology allows organizations to extend their online operations over long distances to form global networks and to support distributed processing. The Pathway environment, in conjunction with the Tandem NonStop Kernel operating system, allows you to distribute application processes within a single system. Additionally, NonStop TS/MP and NonStop TM/MP, in conjunction with the Expand networking software, allow you to spread processes, data, and transactions across a network of Tandem NonStop systems. The coordination of transactions among application servers residing within an Expand network and possibly accessing different resource managers (NonStop SQL/MP and Enscribe) is known as distributed transaction processing (DTP).

Pathway Applications

Pathway applications consist of two types of programs: requester programs and server programs. This design allows application logic to be distributed near the resources it manages. For example, presentation services are located near terminal devices or workstations; database logic resides in server programs near that database. Requesters and servers communicate by using the Guardian file system or the message system that is part of the Tandem NonStop Kernel.

Users interact with your application by using devices and processes controlled by your requester programs. Often these devices are terminals through which the users enter and retrieve transaction data. They might also, however, be intelligent devices such as personal computers, workstations, point-of-sale devices, or automatic teller machines (ATMs). Or, they might be Guardian processes that provide transaction input from a file or other batch medium.

Server processes receive requests from requester processes to access a database to add, retrieve, or modify information. Server processes process request messages and send reply messages with the results of the work on the database.

Servers and Server Classes

You can write Pathway server programs in C, C++, COBOL85, Pascal, the Transaction Application Language (TAL), the Portable Transaction Application Language (pTAL), FORTRAN, or Extended BASIC in the Guardian environment. Alternatively, you can write Pathway server programs in C or COBOL85 in the NonStop Kernel Open System Services (OSS) environment; you must program such servers to read the Guardian $RECEIVE file as described in the *Open System Services Programmer's Guide*. In both cases, you configure and manage the servers using the PATHCOM interactive interface or the Pathway management programming interface (based on the Subsystem Programmatic Interface, or SPI) in the Guardian environment.

The same server programs, whether developed in the Guardian environment or in the OSS environment, can be used with several different requester and client interfaces. These interfaces include SCREEN COBOL, the Pathsend procedures, the Remote Server Call (RSC) interface, and the Pathway Open Environment Toolkit (POET).

The Pathway environment provides the feature of server classes. A server class is a collection of replicated Pathway server processes. All server processes in a server class provide the same set of functions; that is, they execute the same program.

Server Processes

Server processes provide the following benefits:

- Server processes help ensure transaction integrity and, therefore, the integrity of the database.
- Server code can be reused by many requester programs, and you can separate presentation services from database functions.
- You can control which transactions can be performed on your node. You can control the logic of the servers, database names, disk names, and so on.
- In distributed environments, server processes provide high performance by allowing you to use remote servers instead of performing multiple remote I/O operations, placing transaction processing close to system resources.

Server Classes

Server classes provide the following benefits:

- You can minimize use of system resources—for example, processes and file opens—because server classes are shared and highly utilized.
- You can maximize performance because server classes allow multiple copies of server processes to run concurrently in multiple processors.
- Based on configuration settings determined by the system manager or operator, the PATHMON process can dynamically create additional server processes within the server class to maintain acceptable throughput as the workload increases.

- By temporarily freezing and stopping the server class and changing configuration parameters, the system manager or operator can adjust the number of servers that are active at any one time to suit response-time requirements.
- The system manager or operator can balance the workload over multiple processes and across multiple processors, which provides fault tolerance in addition to load balancing: if a processor fails, the server class is still available.

Requesters

The Pathway application programming environment provides two programming interfaces for requesters:

- The Pathsend application program interface (API), provided in the NonStop TS/MP product
- The SCREEN COBOL language, provided in the Pathway/TS product

Requesters written using these two interfaces are briefly described in the following paragraphs. In addition, other Tandem products are available to assist you in writing requesters and clients that communicate with Pathway servers. These products include the Remote Server Call (RSC) product and the Pathway Open Environment Toolkit (POET) for workstation clients and the Extended General Device Support (GDSX) product for front-end and back-end processes.

Section 2, Designing Your Application, provides additional information about how Pathsend requesters, SCREEN COBOL requesters, RSC and POET clients, and GDSX processes can be used in Pathway applications.

Pathsend Requesters

The Pathsend procedure calls and the LINKMON process allow Guardian processes to access Pathway server classes. The Pathsend procedures bring the benefits of Pathway server classes to a wide range of requesters, providing flexibility in application design. They also provide high performance for requesters that do not need a complex, multithreaded interface to terminals or intelligent devices. Finally, they provide support for both context-free and context-sensitive servers.

Pathsend requesters support the following features:

- Use of the TMF subsystem
- Automatic retry of I/O operations to a server process if the primary process of a server process pair fails, through use of the Guardian file system

The Extended General Device Support (GDSX) product provides a set of “pseudo Pathway procedures” that allow you to call Pathsend procedures in the user-supplied part of a GDSX program. A GDSX process can thus function as a Pathsend requester. GDSX processes can communicate with devices by means of a number of data communications protocols, as described in the *Extended General Device Support (GDSX) Manual*.

SCREEN COBOL Requesters

SCREEN COBOL requesters, which are compiled by the SCREEN COBOL compiler and then interpreted and executed by the terminal control process (TCP), provide ease of programming if you need to handle large numbers of terminals or intelligent devices or if you need screen-presentation services. The TCP and the SCREEN COBOL language produce a high-quality, manageable application. The TCP provides multithreading of requesters, fault tolerance, terminal device configuration, and operations management so that you do not need to program these features in your application. Transaction protection through use of the TMF subsystem, with simplified programming, and automatic retry of I/O operations are also provided. SCREEN COBOL requesters are described in the *Pathway/TS TCP and Terminal Programming Guide* and the *Pathway/TS SCREEN COBOL Reference Manual*.

You can use an Extended General Device Support (GDSX) process as a front-end process to the TCP and SCREEN COBOL requesters to communicate with devices not directly supported by the TCP. Use of the GDSX product is described in the *Extended General Device Support (GDSX) Manual*.

The Pathsend Environment

The Pathsend environment includes Pathsend processes and LINKMON processes:

- Pathsend processes, written as part of your application, use Pathsend procedure calls to make requests to server classes.
- LINKMON processes, supplied by Tandem, control communication between Pathsend processes and Pathway server classes.

Pathsend Processes

In writing programs to run as Pathsend processes, you use a set of procedures that are part of the Guardian procedure library. These procedures allow you to send request messages to server processes within a server class and to receive the servers' replies. You can call the Pathsend procedures from programs written in C, C++, COBOL85, Pascal, the Transaction Application Language (TAL), or the Portable Transaction Application Language (pTAL).

Pathsend procedure calls are provided for both context-free and context-sensitive communication with servers. A context-free server accepts a single message from a requester, performs the requested tasks, and issues a single reply to respond to the requester. After the reply message is issued, the server retains no information (context) that can be used in subsequent requests. A context-sensitive server engages in a multiple-message communication, or dialog, with a requester. Between messages, the server retains information (context) pertaining to the dialog.

The use of the Pathsend procedure calls is described in Section 3, Writing Pathsend Requesters, and their syntax is described in Section 5, Pathsend Procedure Call Reference. Design considerations related to context-free and context-sensitive servers are discussed in Section 2, Designing Your Application.

LINKMON Processes

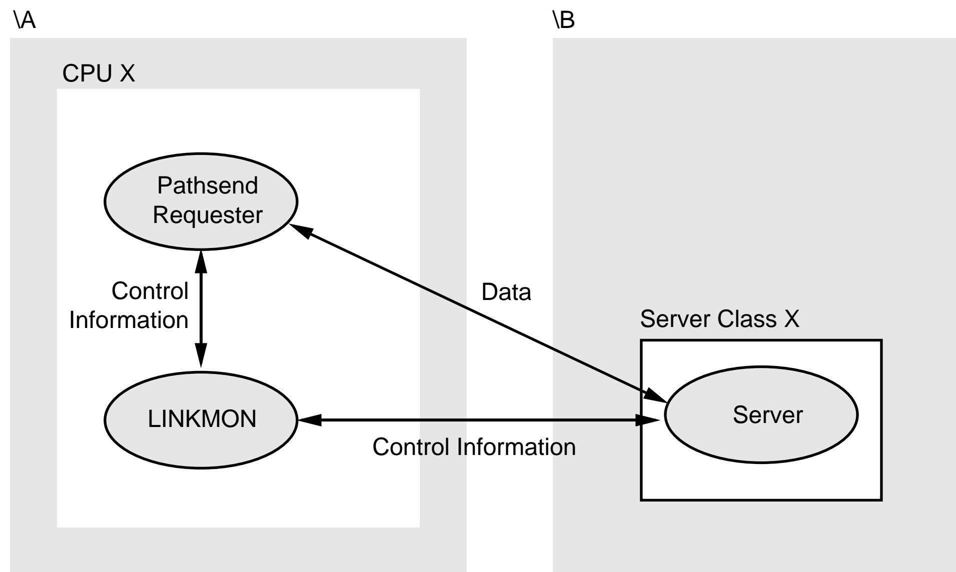
LINKMON processes, together with the PATHMON process, perform link-management functions for Pathsend processes. (A link is a connection to a server process.) A LINKMON process executes in each processor, or CPU, of a system. As a link manager, a LINKMON process is responsible for managing links on behalf of all the Pathsend processes executing in its processor.

If you have the NonStop TS/MP software installed on your system, a LINKMON process is automatically started in each processor. You cannot start a LINKMON process with a RUN command.

Figure 1-1 shows a sample Pathsend environment in which Pathsend processes and a LINKMON process reside in the same processor on system \A. The LINKMON process sets up communication to the Pathway server class on system \B through the PATHMON process controlling the server class. The role of the PATHMON process in establishing this communication is described in Section 3, Writing Pathsend Requesters.

As shown in the figure, only server-class control information is passed to the LINKMON process; the application data moves directly from the Pathsend requester process to the server process.

Figure 1-1. Pathsend Interprocess Communication



Although you can obtain some information about LINKMON processes through the PATHMON process (by means of PATHCOM or SPI), LINKMON processes are not managed as PATHMON-controlled objects. For details about management of LINKMON processes, refer to the *NonStop TS/MP System Management Manual*.

Client/Server Capabilities

The Remote Server Call (RSC) product and the Pathway Open Environment Toolkit (POET) bring client/server capabilities to the Pathway environment by allowing you to move requester functions to a workstation. RSC allows client programs residing on a workstation to access Pathway server classes in any of three different ways:

- Through a Pathsend requester provided by RSC; this requester works with the LINKMON process.
- Through a special intelligent device support (IDS) requester supplied with RSC; this requester works with the terminal control process (TCP) provided in the Pathway/TS product.
- Through an IDS requester that you develop yourself in the SCREEN COBOL language; this requester works with the TCP provided in Pathway/TS.

RSC also allows requesters to access Guardian processes directly. To facilitate access to servers and Guardian processes, RSC consists of multiple components within both the workstation and Tandem computer environments.

The Pathway Open Environment Toolkit (POET) provides tools for developing RSC clients for the Microsoft Windows environment. These tools include a simplified programmatic interface, name mapping, and data conversion mapping.

For information about RSC, refer to the *Remote Server Call (RSC) Programming Manual*. For information about POET, refer to the *Pathway Open Environment Toolkit (POET) Programming Manual*.

Other Transaction Processing Environments

The NonStop TS/MP product serves as the foundation for open transaction processing on Tandem NonStop systems. In addition to the Pathway environment, NonStop TS/MP supports the NonStop TUXEDO transaction processing system. This product allows you to develop TUXEDO transaction processing applications to run on Tandem NonStop systems, thus providing these applications with the fundamental advantages of Tandem NonStop systems.

When using the NonStop TUXEDO system, you work in the NonStop TUXEDO programming environment; you need not use the requester and server programming interfaces described in this manual. Note, however, that you can develop applications that use a combination of modules from the NonStop TUXEDO environment and the Pathway environment.

You can write a Pathsend requester that also acts as a NonStop TUXEDO client, directly invoking the services of a NonStop TUXEDO server, by using the NonStop TUXEDO Application Transaction Monitor Interface (ATMI) functions. For more information about this mechanism, refer to Section 3, Writing Pathsend Requesters, in this manual and to the *NonStop TUXEDO System Application Development Guide*.

Alternatively, you can write a Pathsend or SCREEN COBOL requester that indirectly invokes the services of a NonStop TUXEDO server by using the Pathway to TUXEDO translation server provided with the NonStop TUXEDO product. For more information about this translation server, refer to the *NonStop TUXEDO System Pathway Translation Servers Manual*.

You can write a NonStop TUXEDO native System /T client or OSS workstation client that directly invokes the services of a Pathway server by including calls to the Pathsend procedures described in this manual. For more information, refer to the *NonStop TUXEDO System Application Development Guide*.

Alternatively, you can write a TUXEDO client (or server acting as a client) that indirectly invokes the services of a Pathway server by using the TUXEDO to Pathway translation server provided with the NonStop TUXEDO product. This translation server allows access to Pathway servers from remote TUXEDO requesters (those that use System /Domain) and non-native TUXEDO workstation clients, neither of which can make Pathsend procedure calls. For more information about the TUXEDO to Pathway translation server, refer to the *NonStop TUXEDO System Pathway Translation Servers Manual*.

Development Tools and Utilities

When you are writing requester and server programs for your Pathway application, a variety of program development tools and utilities are available to you. These tools and utilities allow you to shorten the amount of time it takes to code, debug, and test your programs.

Programming Languages and Related Tools

Tandem provides compilers that allow you to write application programs in a number of programming languages, including C, C++, COBOL85, SCREEN COBOL, Pascal, the Transaction Application Language (TAL), the Portable Transaction Application Language (pTAL), FORTRAN, and Extended BASIC. In addition, the Crossref cross-reference generator is available if you want to supplement the cross-reference listings provided by the compilers.

The Inspect Symbolic Debugger

The Inspect product is the symbolic program debugging tool for Tandem NonStop systems. You can use it interactively to examine and modify the execution of Guardian processes (for example, Pathsend requesters and Pathway servers) as well as SCREEN COBOL requesters. An online help facility is available for all Inspect commands and topics.

Using the Inspect product in a Pathway environment requires the use of two terminals or a terminal emulator with windowing capability. One terminal or window acts as the application terminal, while the second terminal or window acts as a command or Inspect terminal.

The Pathmaker Application Generator

The Pathmaker product helps you create Pathway applications consisting of requester programs written in SCREEN COBOL and server programs written in C or COBOL85. To create applications with the Pathmaker product, you:

- Enter information about your application into a series of screen-based entry forms, which the Pathmaker product then stores in a catalog
- Use the Tandem text editor, TEDIT, to create source files containing C or COBOL85 service code

At your command, the Pathmaker product uses the information from the catalog and the TEDIT file to generate SCREEN COBOL requester code, C or COBOL85 server code, and command files to configure and start the finished Pathway environment for testing.

The Pathmaker product simplifies the creation of Pathway applications by:

- Generating application code in a uniform structure for all requesters and servers, to help simplify maintenance and modification
- Producing program statements for tasks that are specific to the Pathway environment

- Automatically generating TMF statements in your requester programs when you indicate that you want your programs to have TMF protection
- Providing a central location for most application information
- Creating error-handling code for the most commonly encountered errors
- Letting you simulate application screens and navigate from one application screen to another before you write a single line of code

Applications developed with the Pathmaker product can access data from databases managed by either the NonStop SQL/MP relational database management system or the Enscribe database record manager. If you are using Pathsend requesters, or clients that use RSC or POET, you can use the Pathmaker tool to create prototype servers.

Client/Server Development Tools

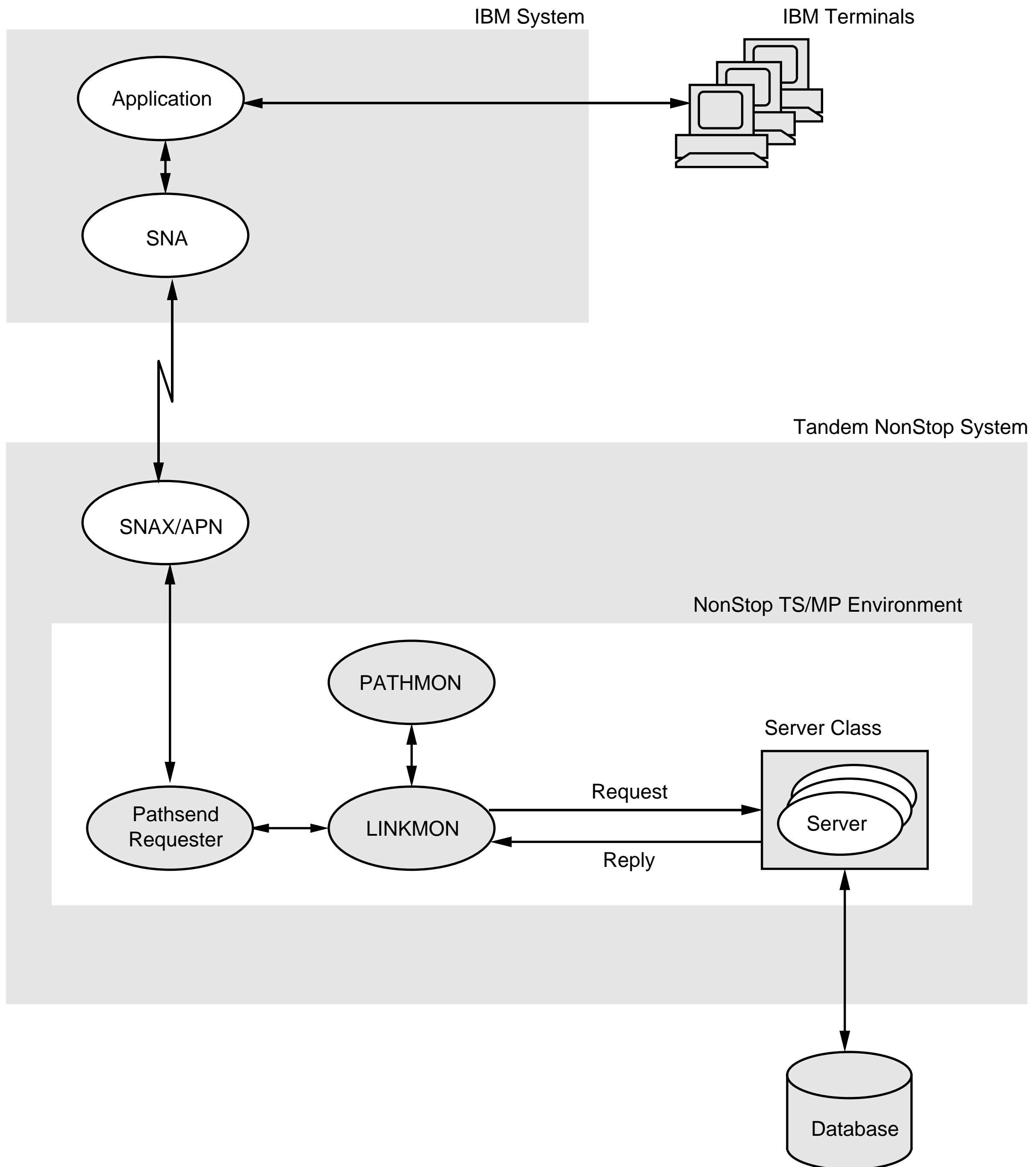
As mentioned earlier, the Remote Server Call (RSC) product facilitates client/server computing, allowing workstation applications to access Pathway servers. A large number of packaged tools and utilities are commercially available for use with RSC, including Tandem’s Pathway Open Environment Toolkit (POET).

Transaction Processing Scenario

Figure 1-2 and the description that follows it provide an example of how transactions from Pathsend requesters are processed. The figure shows the path of transactions from an IBM system to a server class on a Tandem NonStop system.

Note. The figure does not reflect the actual flow of data from the Pathsend requester to the Pathway server class. Only server-class control information is passed to the LINKMON process; the application data moves directly from the Pathsend process to the server class.

Figure 1-2. Example Application Using a Pathsend Requester



In this scenario, clerks at an order entry office enter their transactions into terminals attached to an IBM system. Processing of the transactions, however, requires access to a database that is linked to a Tandem NonStop system.

1. The clerks enter transactions into their terminals and initiate processing by pressing function keys. Any preliminary checking or editing is performed by the application on the IBM system.
2. The IBM system collects the transactions and sends them to a Pathsend requester located on the Tandem NonStop system. The transactions are sent by using a high-speed networking product; for example, Tandem's SNAX Advanced Peer Networking (SNAX/APN) product.
3. The Pathsend requester accepts the transactions for the Tandem NonStop system and formats a request message containing the name of the server class and the data needed by the server to complete its work. The TMF transaction begins.
4. The Pathsend requester forwards the request message to the LINKMON process by calling the Pathsend SERVERCLASS_SEND_ procedure. (This is a context-free message.)
5. If the LINKMON process does not have a link to the specified server class, the LINKMON process asks the PATHMON process for a link to a server process in the server class. The PATHMON process replies that a server process is available. If the LINKMON process already has a link to the server class, this step is not performed.
6. The LINKMON process forwards the request to the server process by using NonStop Kernel interprocess communication.
7. The server process receives and reads the request message.
8. Executing NonStop SQL/MP statements in its program, the server process accesses the database and updates the appropriate information.
9. The server process formats a reply message, which verifies that the information has been updated, and replies to the LINKMON process by using NonStop Kernel interprocess communication.
10. The LINKMON process receives and forwards the reply messages to the Pathsend requester. The TMF transaction ends.
11. The Pathsend requester returns the reply messages to the IBM system, where the application displays the information on the terminal screens.

2 Designing Your Application

To develop a functioning Pathway application, you must identify the individual transactions in your business operations, design and build the application database, and design and code requester programs and server programs. This section describes the design of transactions and databases for Pathway applications and the design of requester and server programs.

To explain these application design tasks, this section uses as an example an application that processes sales orders for a distributorship. The example shows how the Pathway environment can be used to create an OLTP application that supports the distributorship's order-processing operations.

The distributorship in the example has three offices linked by telecommunications:

- \CORP is a network node at corporate headquarters where the purchasing, accounts receivable, and accounts payable functions are managed.
- \WHS is a network node in a warehouse where the inventory, shipping, and receiving functions are performed.
- \REG is a network node in a sales office that is responsible for processing all customer orders in a particular geographic region. Order-processing functions consist of entering orders as input and maintaining records of each order. To perform these two functions, the order processing group:
  - Checks with inventory control to determine if items to be ordered are in stock
  - Sends inventory control shipping and ordered-items information about each order
  - Gets customer credit information from accounts receivable
  - Sends billing information to accounts receivable
  - Answers customer inquiries about order status
  - Records complete information about each order in the database

Designing Transactions

The first step in developing a Pathway application is to identify and define the transactions that your application will process. To do this, you isolate the business tasks you plan to automate, analyze the flow of information within those tasks, list the transactions that result from the analysis, and then identify the various components of the transactions. After these tasks are performed, you protect each transaction, and therefore the integrity and consistency of the database, with the Transaction Management Facility (TMF) subsystem.

Analyzing Data Flow

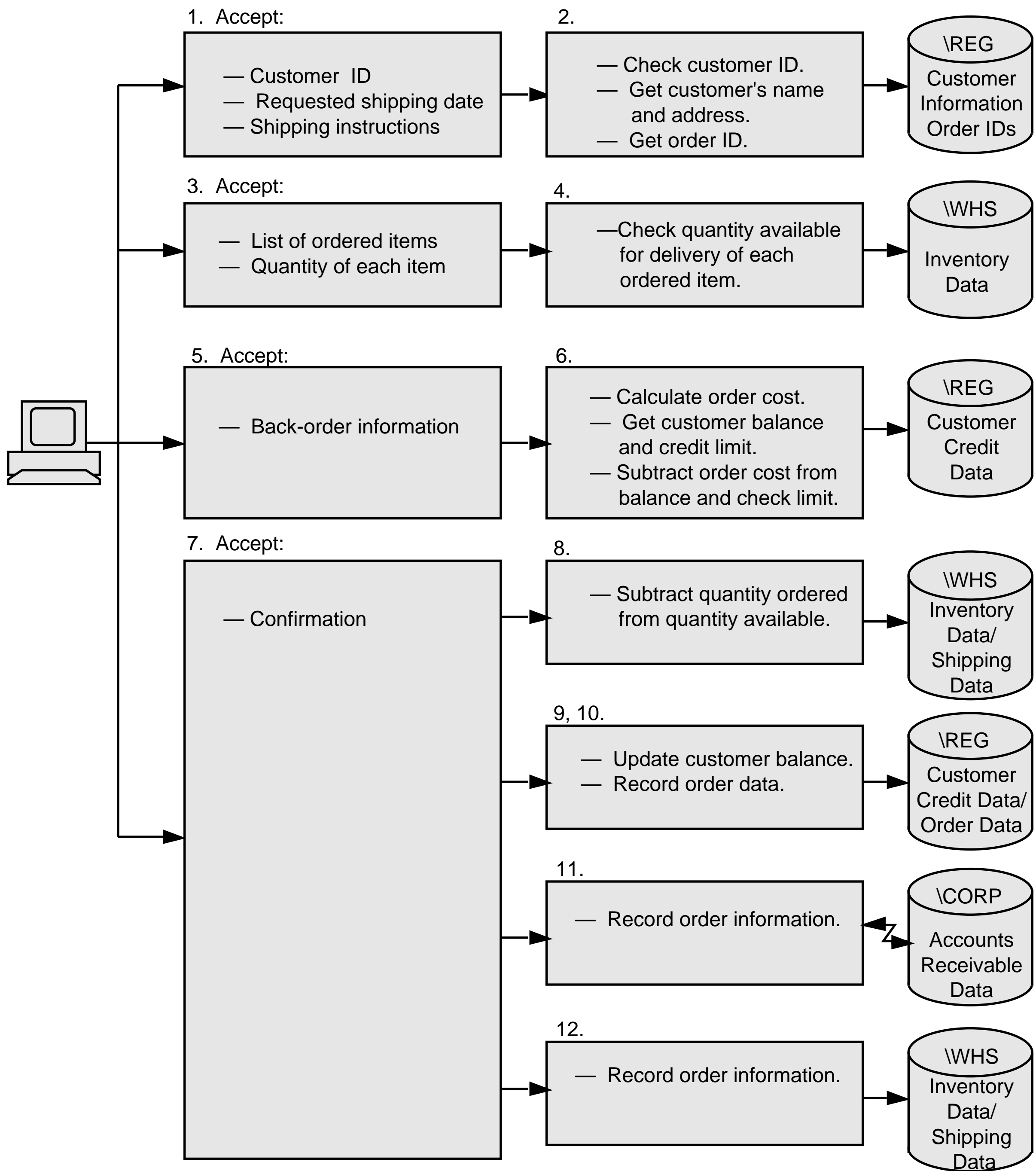
Analyzing the flow of data involves identifying what information is required for a business task, determining the order in which that information is required, and specifying how the information is to be handled. To automate the order-processing tasks of the previously described distributorship, for example, you could analyze the flow of information as follows:

1. Accept the customer's identification number, a requested delivery date for the order, and shipping instructions such as the delivery address.
2. Check the customer's identification number to ensure that the customer is defined in the \REG database; get the customer's name and address from the \REG database; and get a new order identification from the \REG database.
3. Accept a list of order items along with the requested quantity for each order item.
4. Check the current quantity available, in the database on \WHS, of each ordered item to ensure that sufficient quantity exists to fill the order.
5. Accept any special instructions, such as back-ordering out-of-stock items, required to process the order.
6. Calculate the total order cost; get the current customer balance and credit limit from the \REG database; add the total order cost to current customer balance; and ensure that the new balance does not exceed the customer's credit limit.
7. Ask the customer to confirm the order.
8. After the customer has confirmed the order, subtract the quantity ordered from the current quantity available, in the \WHS database, for each ordered item.
9. Add the total order cost to the customer's current balance in the \REG database.
10. Record the order information in the \REG database.
11. Transmit the order information in the accounts receivable files to the \CORP database and record the information in the database.
12. Record the order shipping information in inventory files on the \WHS database.

Assume that your analysis of the previous flow of information shows that only two transactions need to be created to support order processing: an Add New Customers transaction and an Enter Sales transaction. The Enter Sales transaction, which accepts and records all the information associated with a customer order, is the example used in the rest of this section.

The data flow outlined in the previous steps is illustrated in Figure 2-1.

Figure 2-1. Data Flow for a Business Task



Identifying Transaction Components

After you have identified the Enter Sales transaction for the order-processing application, you list the functions performed by the transaction and group them either into data collection and validation operations or into database update operations. For example, the key functions performed by the Enter Sales transaction during data collection and validation are:

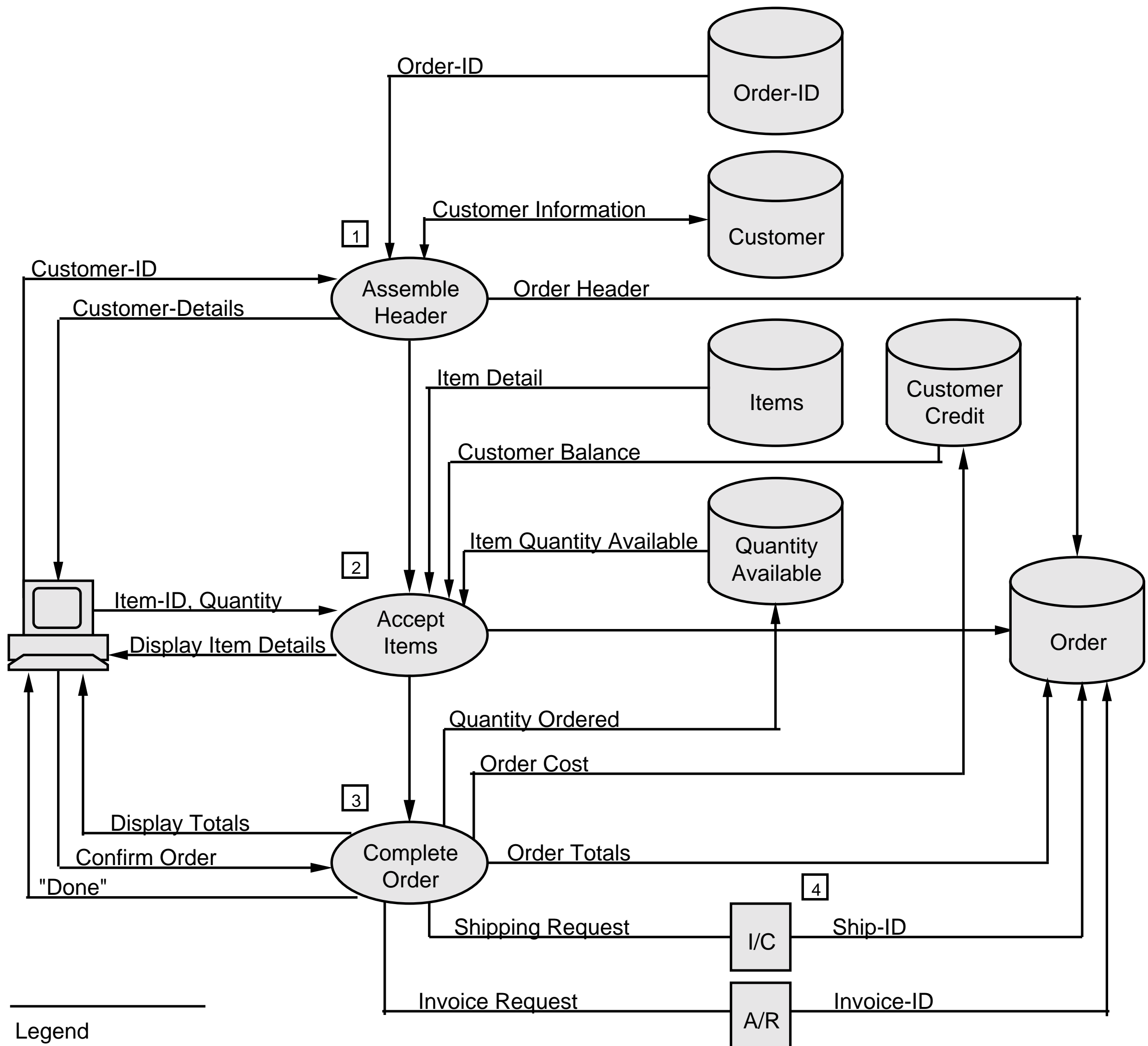
- Assembling information for the order header, including:
  - Obtaining the order-ID
  - Accepting the customer-ID
  - Accepting the requested delivery date
  - Accepting shipping instructions
  - Checking the customer-ID
  - Obtaining the customer’s name and address from the database
- Assembling the order, including:
  - Accepting the list of order items and the quantity of each item
  - Checking the current quantity available for each item ordered
  - Accepting special instructions
  - Calculating total order cost
  - Obtaining the customer’s balance and credit limit from the database
  - Adding the total cost to the customer’s balance and ensuring that it does not exceed the credit limit

The key function performed by the Enter Sales transaction during database update operations is order completion. The order completion function includes:

- Subtracting the quantity ordered from the current quantity available for each ordered item
- Adding the total order cost to the customer’s current account balance
- Recording the order in the database
- Recording the order invoice in the accounts receivable files
- Recording order shipping information in the inventory files

The relationships of the various functions for the Enter Sales transaction are illustrated in Figure 2-2. The dark arrows in the figure show the sequence of actions from Step 1 through Step 3. The lighter arrows show the flow of information.

Figure 2-2. Relationships Between Transaction Functions



Legend

1 Assemble information for order header; display at terminal and add to database; optionally change customer information.

2 Accept items in order, check item availability and customer credit; display item details at terminal and add to database.

3 Display totals at terminal and get confirmation; update item quantity and customer balance; add totals to database; inform related applications about order.

4 Later, when order is shipped and customer billed, add shipping and invoice numbers to database.

CDT029

Protecting Transactions

After listing and grouping the components of the Enter Sales transaction, you protect the integrity of each transaction, and ultimately the consistency of the database, with the TMF subsystem. The following pages outline how to integrate the TMF subsystem with your business transactions. For information about the overall features of the TMF subsystem, including database file recovery and audit trails, refer to the *Introduction to NonStop Transaction Processing*.

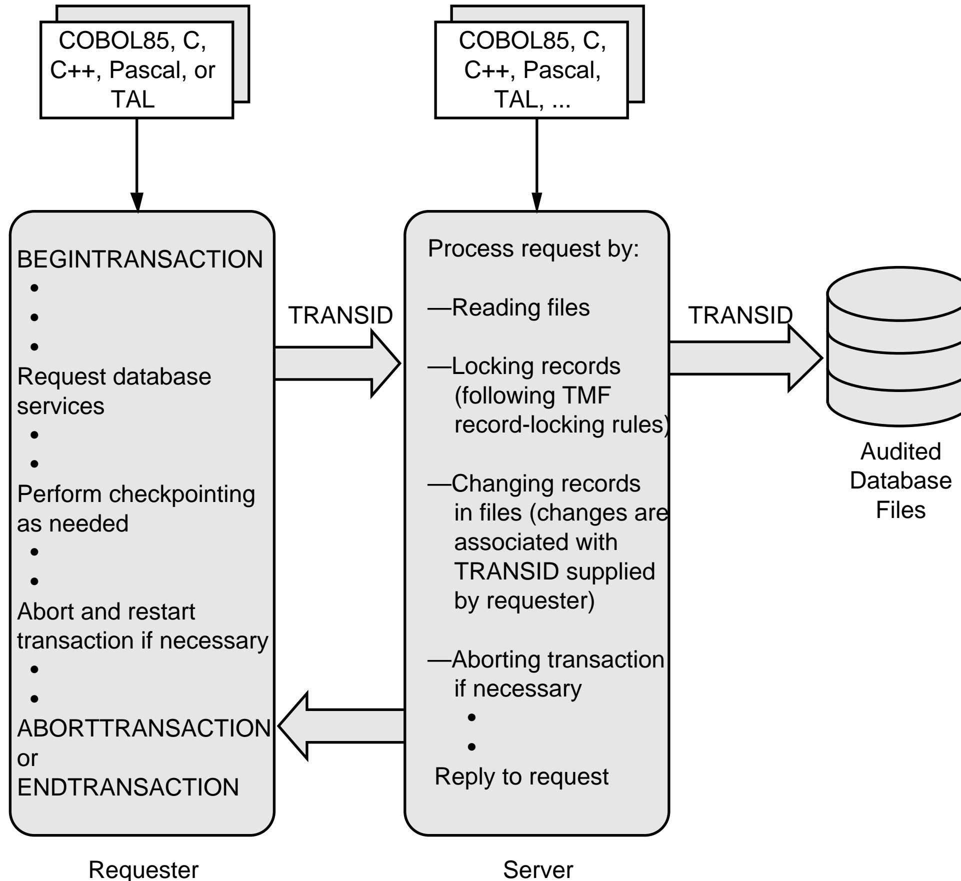
Defining TMF Transactions

From a systems perspective, a transaction includes all the steps necessary to transform a database from one consistent state to another. A TMF transaction must be constructed as a logical unit of work: that is, all parts of a transaction, which usually consists of multiple operations, must be handled as a single entity. If any parts of a TMF transaction are not successfully completed or applied to a database, then none of the transaction parts are applied to the database. By forcing all components of a transaction to be handled as a single unit of work, the TMF subsystem prevents inaccurate or partial updates to the database and protects database consistency.

At the application level, a TMF transaction is defined by special procedure calls or statements that specify the beginning and end of a transaction. For example, in a Pathsend requester program, a transaction begins with a call to the BEGINTRANSACTION procedure and ends with a call to the ENDTRANSACTION or ABORTTRANSACTION procedure. The procedure calls that define TMF transactions act as brackets; that is, the statements are placed before and after the add record, update record, and delete record procedures in your requester program.

Figure 2-3 illustrates the use of the TMF subsystem by a Pathsend requester program and a Pathway server program. In the illustration, the variable called TRANSID acts as a transaction identifier.

Figure 2-3. Pathway Application Programming for the TMF Subsystem



Database Consistency and Concurrency

Potentially, all operations that alter the database are candidates for TMF protection. But before you can apply TMF protection to your transactions, you need to determine:

- When to begin a TMF transaction
- Whether all of the database update operations have to happen together in the same TMF transaction or whether they can be parts of different transactions

To answer these issues, you have to establish your criteria for database consistency and decide how much processing concurrency you can achieve in the application. For example, the Enter Sales transaction affects several pieces of information: order data, inventory data, shipping data, customer credit, and receivables. Upon examination of this transaction, you will see that it is possible to make one general assertion about order processing and about the Enter Sales transaction in particular: An order is not complete until every piece of information associated with the order is recorded in the \REG, \CORP, and \WHS databases.

To illustrate this assertion, consider a situation where a transaction fails after it changes the customer's balance, records the order information, and records the order invoice, but before it records the shipping information. In this scenario, the customer is going to be billed for an order never received. Consequently, your basic criterion for database consistency is as follows: all database updates that are related to the order must be part of one TMF transaction.

Any record modified or inserted by a database operation that is protected by the TMF subsystem is locked and unavailable to other transactions until the initial transaction ends successfully. This type of locking protocol means that you always have a design tradeoff—consistency versus concurrency—with respect to locking records that are actively accessed by the application. If records are locked too early, other transactions cannot access them and the application's concurrency (its ability to process many transactions at the same time) suffers.

As the Enter Sales transaction demonstrates, all of the data collection and validation operations can happen before you begin the TMF transaction—although some revalidation may be done again as part of the transaction. Assembling the order header and assembling the order involve reading records in the database but not changing the records. The rest of the operations change the database and should all be done within a TMF transaction.

As a general rule, you should design the application's transactions to maintain consistency under all circumstances. After the application is installed and running successfully, you can look for ways to improve its concurrency.

Aborting Transactions

If the requester or the server program detects a problem during the processing of a TMF transaction, the requester or server causes the transaction to be aborted with a special procedure call or statement (for example, a call to ABORTTRANSACTION in a Pathsend program). For requesters, the statement that aborts a transaction is executed in lieu of the statement that ends a transaction; for example, in a Pathsend program the requester either completes the transaction with a call to ENDTRANSACTION or causes it to be backed out, because of an error, with a call to ABORTTRANSACTION.

In the past, program designs typically assigned the task of aborting transactions to requesters. Current program design often assigns that task to servers. Servers abort transactions and inform the requesters of those actions, thus ensuring protection of data. The aborting of transactions by servers is described further under Designing Server Programs, later in this section.

The TMF subsystem backs out aborted transactions by using information contained in the TMF audit-trail files. For more information about transaction backout and audit-trail files, refer to the *NonStop TM/MP Application Programmer's Guide*.

Designing the Database

The next step in developing a Pathway application is to design the database that will be accessed and updated by the application. Designing the database, which is a highly specialized activity typically performed by experienced database administrators, involves:

- Precisely identifying the meaning and use of the data as it exists in your business and specifying the database files and records that will store this data. This step is referred to as logical design.
- Choosing file types and keys for the records. This step is referred to as physical design.

In addition to completing a logical and physical design of your database, you must also select a database manager product and ensure that your server programs can interface with that database manager.

Logical Design

During the logical design process, you determine which classes of data must be maintained by your application and identify the relationships that exist between the classes. Each class of data names something that the database will store information about. For example, in an application that processes sales orders, *orders* is a class of data and *order-items* is a relationship between a particular order and the inventory items within the order. These data classes and relationships generally become records in files accessed by the application.

After specifying data classes, you list the attributes (data items) for each class of data. For example, some of the attributes are *order-ID*, *cust-ID*, and *order-total*. These attributes become fields in the records of the database. After specifying attributes for data classes, you diagram the relationships between each of the files in the database and then normalize your database files. To normalize files is to ensure, at a minimum, that:

- There are no repeating fields.
- Data is dependent on the entire key (a unique element) of a field.
- Data is dependent on nothing but the key.

Physical Design

You undertake the physical design of your database by selecting the appropriate file types and record keys for each of the files in the database. Whether you are using the NonStop SQL/MP software or the Enscribe software as your database management system (DBMS), these file types can be classified as key-sequenced, relative, entry-sequenced, or unstructured:

Key-sequenced	Each record in the file has a primary key and up to 255 alternate keys. The primary key is a field or combination of fields within the record.
Relative	Each record in the file has a unique record number, which is the primary key, and can have up to 255 alternate keys. The record number is a unique value that corresponds to the physical location of the record within the file.
Entry-sequenced	Each record in the file has a unique record number and can have up to 255 alternate keys. The record number corresponds to the order in which a record is stored in the file. The primary key is the relative byte address of the record.
Unstructured	Each record in the file has a unique record number that can be used as the primary key. Alternate keys are not supported.

Although the file type you choose depends on your application requirements, generally you should choose key-sequenced files for a database that will be accessed and maintained by a Pathway application. Key-sequenced files provide more flexibility than the other file types.

Database Managers

Databases supporting Pathway applications can run under either the NonStop SQL/MP relational database management system or the Enscribe database record manager. Both of these products support the creation and use of large databases capable of operating in local or distributed systems.

The NonStop SQL/MP (Structured Query Language/MP) product is both a database management system (DBMS) for production environments and a relational database management system (RDBMS) for decision-making in an information-center environment. The NonStop SQL/MP product allows you to think about and represent files in the database as a collection of similarly structured lists. For more information about designing NonStop SQL/MP databases, refer to the *NonStop SQL/MP Reference Manual*.

The Enscribe database record manager provides a record-at-a-time interface between Pathway servers and your database. For more information about designing Enscribe databases, refer to the *Enscribe Programmer's Guide*.

Remote Duplicate Database Facility (RDF)

If disaster recovery of your database is important, the Remote Duplicate Database Facility (RDF) is available to maintain a copy of the database on a remote system. The RDF product monitors database updates audited by the TMF subsystem and applies those updates to the remote copy of the database. For more information about the RDF product, refer to the *Remote Duplicate Database Facility (RDF) System Management Manual*.

Designing Requester Programs

To facilitate the accessing of Pathway server classes from different transaction sources, you can develop requester programs for a Pathway application that use any of the following access approaches:

- SCREEN COBOL and the TCP
- SCREEN COBOL and the TCP with the intelligent device support (IDS) facility
- The Pathsend procedure calls
- The Remote Server Call (RSC) product, with or without the Pathway Open Environment Toolkit (POET)
- The Extended General Device Support (GDSX) product

In Table 2-1, key technical and business considerations are mapped to each way of accessing Pathway servers. More information about each approach is provided following the table.

Table 2-1. Considerations for Requester Programs

Server Access Approach	Large Number of I/O Devices	Support for Intelligent Devices	Multi-Threading Capability	High Perform-ance	Ease of Develop-ment	Fault Toler-ance	TMF Support	Support for Context Sensi-tivity
TCP	X		X		X	X	X	
TCP with IDS	X	X	X		X	X	X	
Pathsend			X	X			X	X
RSC or POET	X	X	X	X	X		X	
GDSX	X	X	X	X		X		X

SCREEN COBOL Requesters

Screen programs for Pathway terminals perform a variety of front-end functions for your Pathway application and are typically written as single-threaded programs in the SCREEN COBOL language. This language offers a simple programming environment and screen-management system to drive Tandem terminals and IBM 3270 terminals. SCREEN COBOL supports both conversational mode (for either block-mode or conversational-mode terminals) and intelligent mode (for intelligent devices and communications lines).

When you write a screen program in SCREEN COBOL, you can take advantage of the features of the Pathway/TS TCP. As supplied by Tandem, the TCP supports:

- Fault tolerance
- TMF transactions
- Multitasking of single-threaded screen programs
- Access to server processes with Pathway server classes
- Unsolicited message processing (UMP)
- System management interfaces (that is, PATHCOM or the Pathway management programming interface)

SCREEN COBOL requester programs do not perform any file I/O operations except to terminals and server classes. A file I/O operation to a server class, which is in the form of a request message, is initiated by the requester program by using the SCREEN COBOL SEND statement.

For information about designing and coding SCREEN COBOL requesters, refer to the *Pathway/TS TCP and Terminal Programming Guide*.

IDS Requesters

Standard SCREEN COBOL requesters are screen oriented; they send data back and forth between the Working-Storage Section of the program and a terminal’s display screen by way of screen templates defined in the Screen Section. Standard SCREEN COBOL requesters use SCREEN COBOL ACCEPT and DISPLAY statements in the Procedure Division to interact with display terminals.

SCREEN COBOL requesters that employ the IDS facility within the TCP send data back and forth between the Working-Storage Section and an intelligent device (or a front-end process that controls the device) by way of message templates defined in the Message Section. IDS requesters use SCREEN COBOL SEND MESSAGE statements and their associated REPLY clauses in the Procedure Division to interact with the intelligent devices or front-end processes.

Although IDS sends and receives data through Message Section templates instead of Screen Section templates, the TCP still provides:

- Link management for access to Pathway server classes
- TMF support to ensure transaction protection and database integrity

- Fault tolerance through process pairs
- Multithreading and multitasking
- Expanded I/O editing support for data streams from intelligent devices

For information about designing and coding IDS requesters, refer to the *Pathway/TS TCP and Terminal Programming Guide*.

Pathsend Requesters

As an alternative to writing SCREEN COBOL requesters, you can write Pathsend requesters in C, C++, COBOL85, Pascal, TAL, or pTAL. In such requesters, you use Pathsend procedure calls to communicate with Pathway servers. The LINKMON process manages links to your server processes on behalf of Pathsend requesters.

Design Considerations

Pathsend requesters are a good choice for your applications if you need to do the following:

- Take a high volume of transactions from a limited number of devices. In this scenario, there are relatively few requester processes, the requesters are busy, and configuration and management is minimal.
- Access servers that are shared by Pathway requesters and applications other than OLTP applications: for example, a security checking server or a logging server. If such servers are used infrequently or if the workload varies, server processes can be automatically deleted when not needed and restarted through the PATHMON process when needed again.
- Access servers from environments containing a mix of online transaction processing and batch processing: that is, environments where the same set of servers handle both online requests and requests from batch applications such as NetBatch Plus processes.
- Write nested servers, which act as requesters by making requests to servers in other server classes, perhaps server classes managed by a different PATHMON process.
- Write context-sensitive servers, which are discussed later in this section under “Designing Server Programs.”

Pathsend procedure calls give you more flexibility than WRITEREAD calls for server-to-server communication. The application gets all the advantages of server classes, including advantages not readily available with WRITEREAD; for example, load balancing, adjusting the number of servers to fit response-time requirements, and configuration and operations management. You can use the Pathsend procedure calls in C, C++, COBOL85, Pascal, TAL, and pTAL programs.

The Pathsend procedures and the LINKMON process, however, do not provide multithreading, fault tolerance, device configuration, or operations management for requesters. Therefore, if you need these capabilities in a Pathsend requester, you must provide the programming for them.

In addition, Pathsend procedure calls that send messages to server classes must be protected by the TMF subsystem to ensure data integrity in your Pathway application.

The Pathsend procedures and the LINKMON process do not support the checkpointing of Guardian interprocess message synchronization IDs. This lack of checkpointing support is an important consideration when writing fault-tolerant requester programs that do not use the TMF subsystem. Section 3, Writing Pathsend Requesters, provides more information about writing fault-tolerant Pathsend programs.

The Pathsend procedures allow you to indicate a specific timeout value for each message sent to a server class. For example, if you perform SERVERCLASS_SEND_ calls to local and remote systems, you can specify shorter timeout values for the local sends and longer values for the remote sends.

You can restrict access to Pathway server classes by Pathsend requesters by having the LINKMON process perform security authorization checks on each send operation.

The Pathsend procedures and the LINKMON process log error messages in the event of a processing failure. Your Pathsend requester can check for these errors and perform recovery actions.

Program Structure

The example in Example 2-1 outlines a Pathsend requester program written in TAL. This program handles data entry for the order-processing application introduced at the beginning of this section.

Note. The program in Example 2-1 illustrates program structure only; it is not a complete program. For an example of a complete, running Pathsend requester program, refer to Appendix B, Examples.

Example 2-1. Sample Pathsend Requester Program Structure

```
                                                    Declare program variables.

TMFError := BEGINTRANSACTION( TransactionTag ):     Begins TMF transaction.
                                                    Allocate buffer for request
                                                    and reply messages.

PathmonName ':=' ["$PM"]                            Set SERVERCLASS_SEND_ parameters.
PathmonNameBytes :=3;
ServerClass ':=' ["ASERVER"]
ServerClassBytes =7;
Timeout := -1D;                                     Specifies no timeout.
Flags := 1;                                         Specifies nowait send.

SendError := SERVERCLASS_SEND_( PathmonName,        Performs send operation.
  PathmonNameBytes, ServerClass, ServerClassBytes,
  RequestBuffer, RequestBufferBytes, MaxReadCount,
  CountRead, Timeout, Flags, SCSendOpNum, Tag);

IF( SendError ) THEN                                Nowait send request failed.
    BEGIN
    InfoError := SERVERCLASS_SEND_INFO_             Gets Pathsend and File errors.
      ( PSError, FSError );                         Perform application-dependent
   .                                                error logic.
   .
   .
    END;

Filenum:=SCSendOpNum
CALL AWAITIOX ( FileNum, Countread, Tag, -1D );      Wait for completion of send
operation
CALL FILEINFO ( FileNum, SendError );                and get the resulting send error.

IF( SendError ) THEN
    BEGIN
    INFOERROR := SERVERCLASS_SEND_INFO_ ( PSError, FSError );
   .
   .
   .
    END;

TMFError :=RESUMETRANSACTION( TransactionTag );      Resumes TMF transaction.

IF( ReplyError ) THEN                                Perform application-dependent
   .                                                 error logic, which may include
   .                                                 aborting transaction.
   .
ELSE
    TMFError :=ENDTRANSACTION;
```

Clients Using RSC and POET

The RSC (Remote Server Call) product facilitates client/server computing, allowing workstation applications to access Pathway server classes and Guardian processes. The RSC product supports a number of different transport protocols and workstation platforms. For detailed information about the supported platforms and protocols, refer to the *Remote Server Call (RSC) Programming Manual*.

Transactions are transmitted from the workstation application (the client) to a Pathway application running on a Tandem NonStop system (the server) by means of a supported communications protocol, such as NETBIOS, TCP/IP, or an asynchronous connection.

RSC includes a process called the Transaction Delivery Process (TDP), which resides on the Tandem NonStop system. The TDP is a multithreaded process that can handle multiple workstations. It routes request messages from workstations to Pathway server

classes by using either the Pathsend API and the LINKMON process or the terminal control process (TCP) provided in the Pathway/TS product. If the TCP is used, it can route a request message to a Pathway server by using either the intelligent device support (IDS) requester supplied as part of RSC or an IDS requester that you develop yourself. The TDP can also send request messages from a workstation to a Guardian process.

The Pathway Open Environment Toolkit (POET) provides tools for developing RSC clients for the Microsoft Windows environment. These tools include a simplified programmatic interface, name mapping, and data conversion mapping.

For information about designing and coding requesters with the RSC product, refer to the *Remote Server Call (RSC) Programming Manual*. For information about using the POET product, refer to the *Pathway Open Environment Toolkit (POET) Programming Manual*.

Requesters Using GDSX

The Extended General Device Support (GDSX) communications subsystem product simplifies the development of front-end processes and back-end processes for communication with I/O devices. These devices can be of any type, including workstations, terminals, ATMs, point-of-sale (POS) devices, and industrial robots. GDSX supplies code that provides multitasking and other features useful for developing these front-end and back-end processes.

A GDSX process can act as a front-end process for LINKMON processes or a Pathway/TS terminal control process (TCP).

A GDSX process contains two primary parts:

- TSCODE, supplied by Tandem
- USCODE, supplied by the application programmer

TSCODE provides generic routines and management services that help you build a multithreaded, fault-tolerant process. TSCODE provides the following functions:

- Creates new tasks and stops tasks
- Receives all system messages and I/O requests
- Dispatches (wakes up and executes) the appropriate active task to process messages and requests
- Handles errors

USCODE consists of user exits that are called by TSCODE to handle the application-specific, data communications-related functions, such as data manipulation, protocol conversion, and message routing for the I/O process. USCODE is typically written in the Transaction Application Language (TAL) or the Portable Transaction Application Language (pTAL) and bound with TSCODE to produce a functional GDSX process.

GDSX provides its own interface to Guardian procedures, NonStop TM/MP procedures, and Pathsend procedures. The names of the GDSX procedures typically look like their Guardian, NonStop TM/MP, or Pathsend equivalents, but they have a circumflex (^)

character inserted before the procedure name. For example, SERVERCLASS_SEND_ becomes ^SERVERCLASS_SEND_. The GDSX interface supports both context-free and context-sensitive Pathsend procedures.

When a GDSX process is used as a front-end process, multiple threads of a user-coded device handler provide separate tasks to manage the input from I/O devices and provide functions such as data-stream conversion, implementation of a communications protocol, and network communications error handling. One instance of the device handler manages one I/O device.

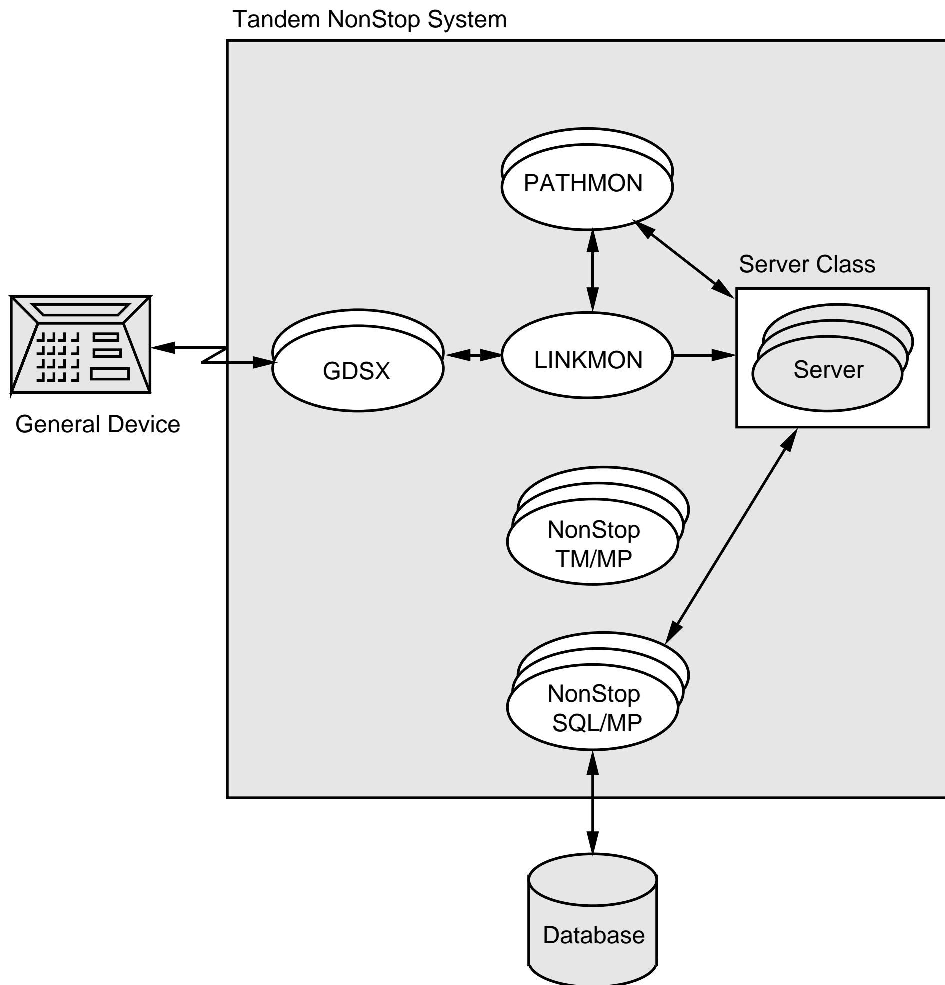
If the GDSX process is acting as a front-end process for a TCP, the GDSX process simulates a terminal supported by the TCP; the simulated terminal is typically run by an IDS requester program. When the IDS facility is used, the GDSX process does not ordinarily control how data appears to the intelligent devices, nor does it perform any other device-dependent functions. However, the GDSX process can be designed to perform device-dependent functions if needed.

A GDSX process can also act as a front-end process to a LINKMON process, as shown in Figure 2-4. The figure shows the path of a transaction from a general device to a Pathway server through a GDSX process.

In this example, the GDSX device handler contains the application requester logic and uses the Pathsend interface to communicate with Pathway servers. Normal interaction with a server process for each thread is similar to that of a Pathsend requester process.

Note. Figure 2-4 does not reflect the actual flow of data from the GDSX processes to the Pathway server class. Only server-class control information is passed to the LINKMON process; the application data moves directly from the GDSX process to the server class.

Figure 2-4. GDSX as a Front-End Process



When developing a front-end process using GDSX, consider the following points:

- A GDSX front-end process is a good choice when a specified data communications protocol is not supported by the Pathway TCP but is supported by GDSX.
- A GDSX front-end process is also a good choice when performance is critical. SCREEN COBOL might not be efficient enough to handle a large amount of application function.
- A GDSX process can be designed to be context-sensitive, through use of the context-sensitive Pathsend pseudo-procedures.
- GDSX processes are managed either through the Subsystem Control Facility (SCF) interactive interface or through a management application program using the Subsystem Programmatic Interface (SPI).

For further information about designing and coding GDSX processes, refer to the *Extended General Device Support (GDSX) Manual*.

Dividing Function Between Requester and Server

In designing a Pathway application, you must decide how to divide function between requester and server. In making this decision, you should consider the type of requester or client you are writing (SCREEN COBOL, Pathsend, RSC, or GDSX), and you should also consider performance, maintainability, and other factors.

For example, which program module should check entry fields for validity? If you are writing a SCREEN COBOL requester, you can easily code it so that the TCP performs these checks. However, a special edit-checking server could provide better performance. If your application includes a workstation requester that communicates with servers using RSC, having the requester check the entry fields would save communications overhead.

As another example, which program module should change screen field attributes such as color, blink, brightness or reverse video for such purposes as highlighting an entry field that contains an error? The SCREEN COBOL language allows such work to be done by the requester, but it could also be done by the server.

For more considerations about dividing function among modules within an application, see Packaging Server Functions, under Designing Server Programs, later in this section.

Designing Server Programs

Request validations, security checks, calculations, database inquiries, and database changes made in response to a request message should be performed by individual units of code within Pathway server programs. As an application programmer, your task is to create a server program to perform specific tasks (for example, create a customer account).

You can write Pathway server programs in C, C++, COBOL85, Pascal, TAL, pTAL, FORTRAN, or Extended BASIC in the Guardian environment. Alternatively, you can write Pathway server programs in C or COBOL85 in the NonStop Kernel Open System Services (OSS) environment; you must program such servers to read the Guardian $RECEIVE file as described in the *Open System Services Programmer’s Guide*. In both cases, you configure and manage the servers using the PATHCOM interactive interface or the Pathway management programming interface (based on the Subsystem Programmatic Interface, or SPI) in the Guardian environment.

Regardless of which operating environment or programming language you use, your Pathway server programs can access database files through the NonStop SQL/MP relational database management system or the Enscribe database record manager. Refer to Designing the Database earlier in this section for information about these two database managers.

You can use the same server programs, whether developed in the Guardian environment or in the OSS environment, with several different requester and client interfaces. These interfaces include SCREEN COBOL, the Pathsend procedures, the Remote Server Call (RSC) interface, and the Pathway Open Environment Toolkit (POET). Requesters or clients using different interfaces can share the same Pathway server classes if you ensure that the server program’s request and reply formats are consistent for all requesters.

After you code and compile your server program, the server object code and library code are shared among all processes of the same server class.

Design Considerations

Before structuring and coding a server program, you should consider some design issues that can affect server performance and efficiency. First, you must decide whether to program single-threaded or multithreaded servers. Additionally, you should be aware of the issues related to context-free versus context-sensitive servers, server packaging, nested servers, aborting transactions, process pairs, early replies, and audited and unaudited servers. You might also consider the use of a GDSX back-end process.

Single-Threaded Versus Multithreaded Servers

When writing Pathway server programs, you need to consider whether to write them as single-threaded or multithreaded programs.

Typically, you can get solid performance from single-threaded servers, which are simpler to design, program, and maintain than multithreaded server programs. For most applications, single-threaded server design is recommended.

Single-threaded servers generally perform well because they are highly utilized. Server processes handle requests from many requester programs (for example, a few servers might support 100 terminals), keeping the server processes highly utilized. Servers are not idle waiting for input from a single requester or device; they can get work from any requester or device.

Low server utilization, however, might still result if the server experiences long waits while processing a request. The main reason for writing multithreaded servers is to provide resource efficiency in applications where processes have long waits (for example, when a server creates Guardian processes on demand or communicates with a remote system from another vendor) or shared access (for example, when requests are multiplexed over a single link).

Context-Free Servers Versus Context-Sensitive Servers

If your servers are programmed to be context-free, the relationship between an application requester program (for example, a SCREEN COBOL program) and a Pathway server process exists only for the duration of a single send operation: that is, one request message and the server's reply to it. Subsequent send operations to the same server class could be serviced by any server process in the server class.

This design allows server processes to be easily shared and serially reused by many requesters. The sharing and reusing of server processes results in more highly utilized servers and consequently can require fewer server processes, depending on the service time required. The assignment of requests to server processes on a per-request basis also allows the PATHMON process to improve the distribution of work among the available servers.

However, context-free design requires that your servers not save any information (that is, context) from previous requests, such as counts, subscripts, or record pointers. Transaction context must be maintained outside of the server, either by the requester

program or in the database. When you program a server to be context-free, you code the server to be independent of its previous request. In essence, every request must be treated as if it were the first request received by the server.

SCREEN COBOL and the TCP support only context-free servers. However, if you use Pathsend requesters, you can use either context-free or context-sensitive servers. In most cases, it is preferable to use context-free servers. However, context-sensitive servers might be a better solution if you need to do one of the following:

- Retrieve large blocks of information (larger than 32 KB) from a database
- Browse a database and repeatedly get the next record, saving the cursor position

Context-sensitive servers require additional programming for both requester and server. The requester must first establish a dialog with a server class and then send messages within the dialog. All messages in a dialog will be sent to the same server process in the server class. Programming details are explained in Section 3, Writing Pathsend Requesters, and Section 4, Writing Pathway Servers.

Packaging Server Functions

Another major decision that you must make during server design is how to package the individual functions, or services, that the server performs. This decision is related to the decision about how to divide function between requester and server, discussed earlier in this section at the end of the subsection Designing Requester Programs.

Based on various operational considerations (for example, security, management, file access, throughput, and response-time requirements), you could choose one or more of the following server packaging alternatives:

- One server class for each database: Called a single-function server, this server (for instance, a list-item server or a change-item-quantity server) services an entire database.
- One server class for each file: Each file has its own dedicated server that executes all I/O (reads, updates, adds, and deletes) against the file. The server performs no functions beyond file access.
- Similar service times: Transactions with similar I/O rates are grouped and processed by the same server. Grouping transactions with different service times in separate servers minimizes server queues and facilitates application tuning.
- One server for each business function: A single server handles all tasks of a specific business function, including business decisions (for example, loan approvals based on predefined criteria) and database navigation.
- One server for similar business functions: Similar business transactions (for instance, enter orders less than $100 in value or delete orders) are handled by a single server.
- One server for similar business functions and all database functions: A single server services similar business transactions and handles both business decisions and database navigation.

- Update or read-only server: A single server exclusively handles either update transactions or inquiry transactions.

One approach to packaging server functions is to first group server functions based on management considerations (for example, all servers within a server class must freeze and stop as a unit) and security considerations (for example, the server class must execute under one user ID). Then, partition server functions based on the database files that are most frequently accessed.

To ensure acceptable response times for users and allow you to tune your application for performance, it is very important to partition server functions based on service times. If the same set of servers handles short and long transactions, some requests for short transactions will be queued behind long transactions, resulting in poor response times for the short requests. If the short and long transactions perform different functions, put those functions in separate server programs. If the short and long transactions perform essentially the same work—for example, a simple database lookup—but some requests could be for multiple lookups, you can configure two or more server classes for requests of different lengths, all using the same program code.

Nested Servers

A server written in C, C++, COBOL85, Pascal, TAL, or pTAL can use the Pathsend procedures to send a request message to a server in another server class and receive a reply. In such a case, the server is acting as a requester. Servers communicating with each other in this manner are called nested servers.

For example, consider a situation where a requester on one node requires the services of two server classes on another node. Instead of sending to server class A, waiting for a reply, and then sending to server class B, the requester could send to server class A, and server class A could send to server class B, get the response, and then reply to the requester. This use of nested servers reduces the number of messages sent across data communications lines and enables application logic to be distributed near the resources it manages.

Consider the following when considering the use of nested server programs:

- Single-threaded servers that send to other server classes can cause process deadlocks. A process deadlock is a situation in which two processes cannot proceed because each is waiting for a reply from the other.

  For example, if a process in server class A sends a request to server class B and the process in server class B then sends a request to server class A, a deadlock might occur. Even if there is more than one process in server class A, there is no guarantee that the second request would not be sent to the same process that sent the original request.

  To avoid this problem, the server program for server class A should keep a read operation posted on $RECEIVE and wait for completion of either the send operation or the read operation. Although this multithreading increases the complexity of the program, it is necessary in order to prevent deadlock.

- Single-threaded servers that send to other server classes can cause low server utilization in the same way that any single-threaded process that calls another process can: the server process sending the request is idle until it receives a reply from the server to which it sent the request.
- Single-threaded servers that send to other server classes can, therefore, result in longer queues for a server class, and these longer queues can affect application performance.

Aborting Transactions

A request sent to the server can have one of three outcomes:

- All the work for the request was completed successfully.
- None of the work for the request was completed.
- The work for the request was only partially completed.

In the first case, the requester can commit the transaction. In the second case, the requester can commit the transaction and then retry it. In both of these cases, the information in the server’s reply is sufficient to ensure the integrity of the transaction.

However, if the transaction work was only partially completed, as in the third case, the server needs to ensure that the transaction is not committed, so that the incomplete work can be backed out. To ensure transaction backout, the server should call the ABORTTRANSACTION procedure after it reads the server’s request and before it sends its reply. A call to ABORTTRANSACTION by the server does not end the transaction—only the requester can end it—but it ensures that the transaction is aborted. The requester should then call either ABORTTRANSACTION or ENDTRANSACTION after it replies to the server. (If the requester calls ENDTRANSACTION in this situation, the ENDTRANSACTION call returns an error because the transaction has already been aborted. However, either call ensures that the resources associated with the transaction are released.)

Fault-Tolerant Process Pairs

You should not write your server processes as fault-tolerant process pairs when you are using the TMF subsystem to protect transactions. The additional programming and functional overhead required to do so is unnecessary.

Early Replies

You should program your servers so that when a server process reads a request message, it completely processes the request before it replies to it. This practice is often called the no-early-reply rule. If you do not follow this rule, the server process loses the TMF transaction identifier it requires to finish processing a transaction. After the reply, any further attempts to lock, write, or delete records in the same audited files will fail. In addition, failure to follow the no-early-reply rule can create a queue of incomplete transactions for single-threaded servers. The queue occurs when the requester, which has been replied to prematurely, sends one more request to the server and the server

increases its potential queue by one request. A single-threaded server queue can result in poorer performance for the application system.

Audited and Nonaudited Servers

If your Pathway application uses a database that is a combination of TMF audited files and nonaudited files, write separate servers to process the two types of files. Updates to audited files must occur within a TMF transaction; updates to nonaudited files should not occur within a transaction, because the transaction imposes unnecessary overhead.

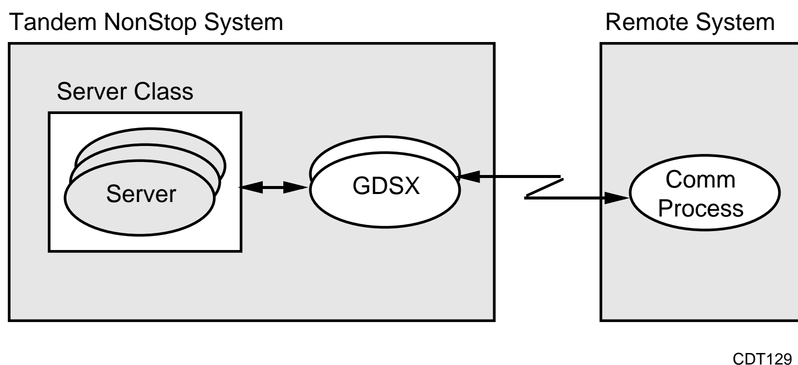
Use of a GDSX Back-End Process

The Extended General Device Support (GDSX) communications subsystem product, described under Requesters Using GDSX, earlier in this section, can also be used as a back-end process for Pathway server classes.

A GDSX back-end process receives input from multiple processes on a Tandem NonStop system and provides access to a limited number of I/O devices. Common uses of GDSX back-end processes include implementing communications protocols; message switching; and coordinating access to shared resources or I/O devices, such as log files, terminals, or remote ports.

In a back-end process, each thread of the device handler has no direct association with an I/O device. Figure 2-5 shows a typical message-switching application in which processes on the Tandem system send messages to the communications process on the remote system. Here, the GDSX process might run a line-handler task to handle data on the communications lines and a device-handler task for each server process. The device handlers forward data to the line handler for forwarding to the remote system.

Figure 2-5. GDSX as a Back-End Process



For further information about designing and coding GDSX processes, refer to the *Extended General Device Support (GDSX) Manual*.

Server Program Structure

Example 2-2 illustrates the structure of a single-function server program written in COBOL85. The program in the following example handles data entry for the order-processing application introduced at the beginning of this section.

Note. The program in Example 2-2 illustrates program structure only; it is not a complete program. For an example of a complete, running Pathway server program, refer to Appendix B, Examples.

Example 2-2. COBOL85 Server Program Example (page 1 of 3)

```
IDENTIFICATION DIVISION.                                Declares server name.

  PROGRAM-ID.     ORDER-SERVER.

ENVIRONMENTAL DIVISION.

  CONFIGURATION SECTION.
   SPECIAL NAMES.                                       Defines source library file
    FILE "$src.srvlib.orderlib"   IS COBOL-LIB.         names.
    FILE "$obj.srvlib.ocmgrobj"   IS COMM-MGR.

  INPUT-OUTPUT SECTION.                                 Selects logical names for
   FILE CONTROL.                                        $RECEIVE and all database
    SELECT MSG-IN                                       files accessed by server.
     ASSIGN TO $RECEIVE                                 $RECEIVE is a Guardian
     FILE STATUS IS RCV-STAT IN WS-RCV-INFO.            file that receives and stores
                                                        messages transmitted between
    SELECT MSG-OUT                                      requesters and servers.
     ASSIGN TO $RECEIVE
     FILE STATUS IS RCV-STAT IN WS-RCV-INFO.

    SELECT LAST-ID
     ASSIGN TO $DATA.ORDER.LASTID
     ORGANIZATION IS RELATIVE
     ACCESS IS SEQUENTIAL
     RECORD KEY IS WS-LASTID-REL-KEY
     FILE STATUS IS FILE-STAT IN WS-FILE-INFO.
       .
       .
       .

    RECEIVE-CONTROL.                                    Declares number of
     TABLE OCCURS 5 TIMES                               concurrent opens and
     SYNCDEPTH IS 1                                     maximum size of largest
     REPLY CONTAINS 204 CHARACTERS.                     request message.

DATA DIVISION.

  FILE SECTION.                                         Declares record data
   FD MSG-IN                                            structures for logical file
     RECORD CONTAINS 204 CHARACTERS                     names.
     LABEL RECORDS ARE OMITTED

    01 ORDER-MSG.
     .
     .
     .
```

Example 2-2. COBOL85 Server Program Example (page 2 of 3)

```
   FD MSG-OUT
     RECORD CONTAINS 36 TO 204 CHARACTERS
     LABEL RECORDS ARE OMITTED

    01 ORDER-REPLY.
     .
     .
     .
    01 ERROR-STATUS-REPLY.
     .
     .
     .
   FD LAST-ID
     RECORD CONTAINS 12 CHARACTERS
     LABEL RECORDS ARE OMITTED

    01 LAST-ID-RECORD.
       02 LAST-ID       PIC 9(12).
          .
          .
          .
```

WORKING STORAGE SECTION. — *Declares data structures of variables used by server.*

PROCEDURE DIVISION.

DECLARATIVES. — *Declares error procedures to be used when an I/O statement returns an error.*

MAIN SECTION. — *Contains main program logic. Server program begins and ends here.*

```
   PERFORM START.
   PERFORM PROCESS-REQUEST
    UNTIL last-requester-close.
   PERFORM STOP.
   STOP RUN.
```

START SECTION. — *Contains logic that opens all files used by server.*

```
   OPEN INPUT msg-in.
   OPEN OUTPUT msg-out SYNCDEPTH 1.
   OPEN I/O last-id SHARED SYNCDEPTH 1.
    .
    .
    .
```

Example 2-2. COBOL85 Server Program Example (page 3 of 3)

```
PROCESS-REQUEST SECTION.                               Contains logic that reads
 PERFORM GET-MSG IN RCV-MGR.                           requests in $RECEIVE, services
 IF NOT last-requester-close                           requests, and replies to
  PERFORM DO-REQUEST.                                  requests.

 DO-REQUEST.
  IF       function-code OF
            order-check-msg = ORDER-CHECK
           PERFORM DO-ORDER-CHECK
  ELSE IF  function-code OF
            order-check-msg = ORDER-COMIT
           PERFORM DO-ORDER-COMIT
  ELSE     PERFORM BAD-REQUEST IN ERROR-MGR.

 DO-ORDER CHECK.
     .
     .
     .

RCV-MGR SECTION.                                       Provides $RECEIVE I/O
                                                       services.

DB-MGR SECTION.                                        Provides disk-file I/O
                                                       services.

ERROR-MGR SECTION.                                     Provides error-processing
                                                       services.

STOP SECTION.                                          Contains logic that closes
 CLOSE msg-in.                                         all files used by server.
 CLOSE msg-out.
 CLOSE last-id.
   .
   .
   .
```

Designing Applications for Batch Processing

If your Pathway application includes batch processing, consider the different needs of this type of processing in your design.

For example, you might code a Pathsend program that takes its input from a file rather than from a terminal, then sends requests to a server to make updates to a database. This program could be configured as a server, thus operating as a nested server. Its input file might be TMF protected, and the Pathsend program might make updates to it.

An application that does several updates to a database, with each update coded as a separate TMF transaction, could be slow when it performs these updates as a batch job rather than performing them online. For batch processing, it is usually faster to group a number of updates in a single transaction. However, if your batch jobs are very large, note that you should not try to group more than about one thousand updates in one TMF transaction.

3 Writing Pathsend Requesters

This section explains how to write programs that use Pathsend procedure calls to make requests to Pathway servers. These programs can be either one of the following:

- Standard requesters—programs that initiate application requests
- Nested servers—servers that act as requesters by making requests to servers in other server classes

Nested servers are described further in Section 4, Writing Pathway Servers. However, this section (Section 3) describes the use of Pathsend procedure calls to perform requester functions, whether by a standard requester or a nested server. The terms Pathsend program and Pathsend process are used to refer to any program or process that uses Pathsend calls, whether it is a standard requester or a nested server.

Pathsend programs are Guardian requesters, as described in the *Guardian Programmer's Guide*. You can write Pathsend programs in C, C++, COBOL85, Pascal, pTAL, or TAL. You should be familiar with the Guardian requester/server model as implemented in the programming language you are using.

In addition, if you are using the Transaction Management Facility (TMF) subsystem, you should be familiar with the programming guidelines and considerations for TMF requesters, as described in the *NonStop TM/MP Application Programmer's Guide*. Note that the *NonStop TM/MP Application Programmer's Guide* describes requesters that use calls to Guardian procedures rather than Pathsend procedures. In Pathsend requesters that use the TMF subsystem, calls to Pathsend procedures are used instead of calls to WRITEREAD and associated Guardian procedures, and there are other differences as described in this section.

An example of a Pathsend requester program that is also a nested server is given in Example B-1 on page B-2.

The Pathsend Procedure Calls

Pathsend programs use six procedures that are part of the Guardian procedure library:

SERVERCLASS_SEND_
SERVERCLASS_SEND_INFO_
SERVERCLASS_DIALOG_ABORT_
SERVERCLASS_DIALOG_BEGIN_
SERVERCLASS_DIALOG_END_
SERVERCLASS_DIALOG_SEND_

Later parts of this section describe how to use these procedures in context-free and context-sensitive Pathsend programs. Section 5, Pathsend Procedure Call Reference, gives detailed syntax and usage considerations for the procedures.

Interprocess Communication in the Pathsend Environment

Communication between requesters and servers in the Pathsend environment differs from communication between Guardian requesters and servers. Rather than directly opening a particular server process, the requester opens the LINKMON process, which in turn opens a server process selected by the PATHMON process. The LINKMON process's open of the server process is shared among all requesters running in the same processor as the LINKMON process.

The communication begins as follows:

1. The requester calls a Pathsend procedure (for example, SERVERCLASS_SEND_) to request a server-class send operation.
2. The request goes to the LINKMON process running in the same processor as the Pathsend requester process.
3. The LINKMON process checks for a link to a server process in the server class specified in the request. If the LINKMON process has no available link to the server class, the actions performed depend on the settings of the PATHMON configuration parameters NUMSTATIC and MAXLINKS. The following steps might occur:
   a. The LINKMON process sends a "get-link" request to the PATHMON process that manages the server class specified in the procedure call.
   b. The PATHMON process selects a server process in the specified server class. If necessary, it starts and initializes a new server process in the server class.
   c. The PATHMON process sends back to the LINKMON process the Guardian process name of a server process within the specified server class.
   d. The LINKMON process opens the server process of that name. This open is shared among all the Pathsend requester processes running in the same processor as that LINKMON process; therefore, a close operation does not necessarily occur when the communication with a particular requester process is finished.

   For more information about the NUMSTATIC and MAXLINKS parameters and their effects on get-link requests, refer to the *NonStop TS/MP System Management Manual*.
4. The LINKMON process forwards the send request to the server process.
5. When the server process replies, the LINKMON process replies to the requester process, and the server-class send operation is complete.

The action of the PATHMON process in step 3 is called granting a link. The LINKMON process, which requests links and provides access to the server process after the link is granted, is called the link manager. (For SCREEN COBOL requesters, the terminal control process (TCP) serves as the link manager.)

Basic Pathsend Programming

The simplest type of Pathsend program is a context-free program. This subsection provides the information you need to write a context-free Pathsend program. It also provides information common to both context-free and context-sensitive programming. The next subsection provides information about the additional tasks required of a context-sensitive Pathsend program.

Context-free Pathsend programming consists simply of sending messages to a server in a server class and receiving the replies. The sending of a message to a Pathway server class is called a server-class send operation. In a context-free program, only two Pathsend procedures are used: SERVERCLASS_SEND_ to send the messages (and to receive the replies if in waited mode) and SERVERCLASS_SEND_INFO_ to get additional information about the results of the last SERVERCLASS_SEND_ call if the call failed.

The SERVERCLASS_SEND_ procedure enables Pathsend requesters to send data to and receive replies from a specified Pathway server class. This procedure communicates with the LINKMON process in the processor where the Pathsend requester is running, passing information that enables the LINKMON process to choose a link to a server process. After a link to the server has been obtained, the requester's data is sent directly from the requester's data space to the server.

Subsequent server-class send operations to the same server class might not be sent to the same server process; therefore, the requester must provide all necessary context on each send.

When a server process replies to the request, the LINKMON process completes the server-class send operation. If you use waited I/O, the reply data is in the reply buffer when the SERVERCLASS_SEND_ call is completed. If you use nowait I/O, you complete the I/O operation with a call to the AWAITIOX procedure.

A Pathsend requester can use the SERVERCLASS_SEND_INFO_ procedure to get detailed information about server-class send initiation and completion errors.

Programming for Failure Recovery

A Pathsend requester must provide a mechanism for failure recovery. It can do so in one or both of the following ways:

- By using TMF level recovery through the Transaction Management Facility (TMF) subsystem
- By providing fault tolerance through process pairs and checkpointing

TMF level recovery is the recovery of the database to a consistent state through the use of the TMF subsystem. When a failure occurs, the TMF subsystem allows the application to back out (abort) the entire transaction, returning the contents of the database to the values it held when the transaction was started. The application can then retry the transaction.

In general, TMF level recovery is recommended for use with Pathsend programs: this method is easier and faster to program than process pairs, and it handles nonretryable

requests more smoothly. However, because the TMF subsystem guarantees only the consistency of the database and not fault tolerance for other operations (such as messages sent over data communications lines), your application might need to use process pairs along with TMF level recovery.

TMF programming is described in the *NonStop TM/MP Application Programmer's Guide*. The *Guardian Programmer's Guide* discusses process pairs and checkpointing. The following paragraphs provide information specific to the use of these features in a Pathsend program.

Server Process Failures

A server process can fail for reasons that include the following: the server process calls ABEND, a processor fails, or someone stops the process.

If the server process fails while there are outstanding SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ calls to it, error 904 (FEScServerLinkConnect) is returned to the requester. If the server-class send operation is protected by a TMF transaction, the requester should abort the transaction and reissue the request or the dialog. If the request is repeatable, the requester should just retry the request. Context-free requesters need only retry the call to SERVERCLASS_SEND_, but context-sensitive requesters must retry the entire dialog.

If the server process fails while there are no outstanding SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ calls to it but a dialog is in progress, the dialog is aborted and the requester receives error 929 (FEScDialogAborted) on its next call to SERVERCLASS_DIALOG_SEND_.

If the server process is in transaction mode and fails, the TMF transaction is automatically aborted. This treatment of the TMF transaction is a feature of the TMF subsystem.

LINKMON Limit Errors

In some cases, you can recover from a LINKMON limit error by retrying the Pathsend procedure call. Whether a retry will work depends on the design and operating environment of your application, including the configuration of static and dynamic links. Static links between a LINKMON process and a PATHMON process generally persist for some time, depending on the application and the system workload. Dynamic links and server classes come and go more frequently, again depending on the application and the system workload. The number of concurrent server-class send operations is very dynamic.

Therefore, it might be appropriate to retry a call to SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ (after a short wait) if the concurrent calls limit is exceeded. Conversely, it might not be appropriate to retry a call if the limit for the maximum number of PATHMON processes is exceeded.

For more information about PATHMON configuration and performance, refer to the *NonStop TS/MP System Management Manual*.

Pathsend Programming and the TMF Subsystem

The Pathsend procedure calls support TMF transactions. Calling a Pathsend procedure to start a server-class send operation propagates the current transaction identifier, if any, to the server process. If the Pathsend procedure call fails, use of the TMF subsystem allows your application to abort the transaction and retry the call. As an application developer, you do not have to be concerned about the role of LINKMON processes in the propagation of transaction identifiers.

If you use the TMF subsystem, you should check for errors after each call to the BEGINTRANSACTION, ENDTRANSACTION, ABORTTRANSACTION, and RESUMETRANSACTION procedures. Failure to perform these checks could cause important parts of your application to fail. If a server error occurs during a TMF transaction, the requester should explicitly abort the transaction (even though in some cases, the transaction might already have been aborted). For further information, including a list of the file-system errors that can be returned by each of the TMF procedure calls, refer to the *NonStop TM/MP Application Programmer's Guide*.

When designing and coding your application, you should pay attention to whether or not your Pathsend requests are in transaction mode. For example, your application should not be in transaction mode while using Pathsend procedure calls to make requests to subsystems whose operations are not TMF protected, such as the spooler.

Instead, do the following:

1. Save the value of the transaction identifier.
2. Suspend the transaction by calling the RESUMETRANSACTION procedure with a tag value of 0.
3. Make the subsystem request.
4. Resume the transaction by calling the RESUMETRANSACTION procedure with a tag value equal to the saved transaction identifier.

Attempting to send to a server class configured with the TMF parameter set to OFF fails with Pathsend error 917 (FEScServerClassTmfViolation) if the requesting process has a current transaction. The *NonStop TS/MP System Management Manual* describes how to set the TMF parameter in the PATHCOM SET SERVER command.

Fault-Tolerant Programming

The following paragraphs describe issues related to the use of Pathsend procedure calls in fault-tolerant programs and discuss the appropriate levels of recovery that your application can perform after takeover by a backup Pathsend process.

For Pathsend calls that are protected by the TMF subsystem, on takeover the new primary process must determine whether the current transaction ended or aborted. If the transaction aborted, the new primary process can retry the entire TMF transaction. That is, the backup process can begin a new TMF transaction and reissue the Pathsend calls. This retry mechanism relies on the TMF subsystem's capability to back out all work that the server process carried out on the original request.

For Pathsend calls not protected by the TMF subsystem, the proper recovery depends on the nature of the request:

- Retryable requests that are not protected by the TMF subsystem can be repeated many times without adverse effect. An example of this kind of request is a request to read a bank account balance. Requests to retrieve data from a database are retryable requests.

  For these requests, on backup takeover, the backup can simply reissue the request. The request could be processed more than once by different server processes without resulting in data corruption.

- Nonretryable requests that are not protected by the TMF subsystem cannot be processed more than once without having adverse effects. An example of this kind of request is a request to subtract $50.00 from a bank account balance.

  For these requests, there is no way for the server class to detect duplicate requests; Pathsend does not support checkpointing of Guardian sync IDs. Therefore, the backup process cannot send the request again because the operation might be processed more than once. Because the request cannot be safely retried, the Pathsend process cannot ensure that the request gets processed at least once.

Because the request thread suspends while a checkpoint is in progress, checkpointing large buffers can affect the performance of your application. You should checkpoint the entire context of nonretryable requests, but avoid checkpointing unnecessary data: for example, data from retryable requests or data that has not changed since the last checkpoint.

The LINKMON process opens servers that are configured with the TMF parameter OFF with a sync depth of 1, and I/O operations to the server process are automatically retried if the primary process of a server process pair fails.

See the *Guardian Programmer's Guide* for detailed information about checkpointing and sync IDs.

Security Issues

There are two levels of security to consider for Pathsend processes: security at the network level and security at the server-class level. In addition, if you are using the Remote Server Call (RSC) product, you can provide additional security to control access to servers.

Network Security

If your Pathsend process is to access a Pathway server class on another system, the user ID of the PATHMON process controlling the server class has to have corresponding user IDs and remote passwords with the following systems:

- The system where the requesting process is running
- The system where the PATHMON process is running
- The system where the server class is running

This level of security is required because the LINKMON process must be able to open the PATHMON process (to make link requests); the LINKMON process must be able to open the server processes (to send user requests); and the PATHMON process must be able to open the server processes (to send startup messages). All of these opens are performed with the PATHMON user ID.

Note. The user ID of the Pathsend process need not have remote passwords to the PATHMON system or to the server-class system to access the server class. Moreover, the Pathsend-process user ID need not be known on the PATHMON or server-class systems.

Server-Class Security

LINKMON processes perform authorization checks on each server-class send operation to make sure that the user ID of the Pathsend process at the time of the send conforms to the server class's OWNER and SECURITY attributes. You set these attributes for server classes at configuration time if those server classes are to be accessed by Pathsend processes.

The *NonStop TS/MP System Management Manual* describes how to set the SERVER OWNER and SERVER SECURITY parameters in PATHCOM.

RSC Client Security

Remote Server Call (RSC) workstation clients can be allowed or disallowed to communicate with specified Pathway servers. You can set up security validation by creating an Access Control Server (ACS). For more information about creating an ACS, refer to the *Remote Server Call (RSC) Programming Manual*.

Avoiding Coded PATHMON Names

SCREEN COBOL requesters can send requests to Pathway server classes without having to specify the name of the PATHMON process controlling the server class, because the TCP has a default PATHMON process to send to. Pathsend processes, however, must specify the PATHMON name of the server class to send to, because the Pathsend procedures provide no default PATHMON name.

It is possible, however, to avoid coding PATHMON names in Pathsend programs. For example, you can use ASSIGNs containing the PATHMON system and process name. Or, if the Pathsend process is a Pathway server, the process can use the name of its creator as the default PATHMON name to send to. This method of avoiding coding the PATHMON name is reliable as long as the sending server is not associative, in which case its creator might not be a PATHMON process.

The Pathsend program examples BREQ and PATHSRV in Appendix B, Examples, use ASSIGNs to avoid coding PATHMON names. The PATHSRV example also uses the creator default method just described to avoid coding the PATHMON names.

Context-Sensitive Pathsend Programming

If you are writing a context-sensitive Pathsend program, you must follow the guidelines in the previous subsection, Basic Pathsend Programming, and also perform additional programming tasks. This subsection describes these additional tasks and other considerations for context-sensitive programming.

Context-sensitive Pathsend programming involves establishing a dialog between a requester and a server process in a server class, and then sending messages within the dialog. After the dialog is established, the same server process is used for all the messages in the dialog; therefore, the server can retain context between send operations.

A requester starts a dialog by calling SERVERCLASS_DIALOG_BEGIN_. This procedure returns a dialog identifier to be used on subsequent server-class send operations, which are made by calling SERVERCLASS_DIALOG_SEND_. The requester can use multiple calls to SERVERCLASS_DIALOG_BEGIN_, and the resulting dialog identifiers, to engage in multiple simultaneous dialogs. As in the context-free case, the requester can call SERVERCLASS_SEND_INFO_ after a server-class send operation to get detailed information about send initiation and completion errors.

Either the requester or the server can abort the dialog, but only the server can end it. (The server aborts the dialog by returning file-system error 1 (FEEOF) in its reply; it ends the dialog by returning file-system error 0 (FEOK).) To abort the dialog, the requester calls SERVERCLASS_DIALOG_ABORT_. The requester calls SERVERCLASS_DIALOG_END_ to clean up resources after the server has ended or aborted the dialog.

As in context-free programming, the requester can perform context-sensitive server-class send operations either waited or nowait. The requester receives an error indication if the server process has terminated or if it has ended or aborted the dialog.

To participate in a dialog with a context-sensitive Pathway requester, a server must perform additional tasks besides those required of all servers. These additional tasks are described under Writing Context-Sensitive Servers in Section 4.

Using Context-Sensitive Requesters With Context-Free Servers

Context-sensitive requesters can perform single-send dialogs with Pathway servers that are coded to be context-free. However, if these servers check for system messages and they use the Common Run-Time Environment (CRE), they must be modified to recognize and respond to Pathsend dialog abort system messages. Section 4, Writing Pathway Servers, describes how to code servers to handle these system messages.

Resource Utilization

On a requester's first SERVERCLASS_DIALOG_BEGIN_ call, an extended segment is added to the requester's segment space for use as a workspace. This segment can be up to 64 KB in size. It is deallocated when the requester process is terminated.

Programming for Failure Recovery

The SERVERCLASS_DIALOG_BEGIN_ procedure allows two types of TMF protection for dialogs:

- The one-transaction-per-dialog model
- The any-transaction-per-dialog model

You use bit 14 of the `flags` parameter in the SERVERCLASS_DIALOG_BEGIN_ procedure call to select which model will be used.

A value of 0 for bit 14 of `flags` (the default) selects the one-transaction-per-dialog model. In this model, the dialog records the transaction identifier that is current at the time of the SERVERCLASS_DIALOG_BEGIN_ call. Subsequent Pathsend calls for this dialog must use this transaction identifier, or the calls will fail; that is, the current transaction at the time of all Pathsend calls for the dialog must be the same as when the dialog was started.

With the one-transaction-per-dialog model, the transaction cannot commit—that is, calls to ENDTRANSACTION will fail—until the server ends (not aborts) the dialog and the requester calls SERVERCLASS_DIALOG_END_. In other words, when bit 14 of `flags` is set to 0, the TMF subsystem treats a dialog like an I/O operation: the ENDTRANSACTION operation fails until the dialog has finished. The same restriction applies to nested servers: if a server receives a message in a dialog and then initiates a dialog with another server, it must complete the entire initiated dialog before replying to the message from the received dialog.

A value of 1 for bit 14 of `flags` selects the any-transaction-per-dialog model. In this model, all server-class send operations within the dialog will contain the transaction identifier that is current at the time of the send, and there are no restrictions on ENDTRANSACTION other than those associated with calls to the WRITEREAD procedure (as described in the *NonStop TM/MP Application Programmer's Guide*).

When the one-transaction-per-dialog model is used, the dialog must be ended rather than aborted for the transaction to be completed successfully. The requester can ask the server to end the dialog by specifying a user-defined END REQUEST command that is recognized by the server.

If the dialog is aborted, the requester receives error 233, FEScError, and SERVERCLASS_SEND_INFO_ returns Pathsend error 929, FEScDialogAborted. The file-system error returned by SERVERCLASS_SEND_INFO_ provides information about the reason for the abort. If the server explicitly aborts the dialog, the file-system error is FEEOF (1).

If the server abends and the dialog is not in an ended state, the dialog is aborted, and the requester receives error 233, FEScError, and SERVERCLASS_SEND_INFO_ returns Pathsend error 929, FEScDialogAborted.

When the one-transaction-per-dialog model is used, the current TMF transaction, if any, is automatically aborted when the server returns an error value (FEEOF, or any other value besides FEOK or FEContinue) in the reply. When the any-transaction-per-dialog model is used (when bit 14 of the dialog flags is equal to 1), no automatic transaction

abort occurs. In either case, the requester should abort the transaction when it receives an error reply.

If the server is shut down by the operator, the following occurs:

- For dialogs that currently have an I/O operation outstanding, the I/O operation is first completed, and action is then taken depending on the error value returned:
    - If the I/O operation is completed with an error value of FEOK (0), then the FEOK value is passed back to the requester along with all user data.
    - If the I/O operation is completed with an error value of FEContinue (70), the requester receives error 233, FEScError, and SERVERCLASS_SEND_INFO_ returns Pathsend error 929, FEScDialogAborted. No user data is passed back to the requester.
    - If the I/O operation is completed with any other file-system error, the requester receives error 233, FEScError, and SERVERCLASS_SEND_INFO_ returns the file-system error that occurred. No user data is passed back to the requester.

  After completion of the I/O operation, the dialog is aborted and the LINKMON process closes the server process. The server process should interpret the close operation as a dialog abort.
- For dialogs that have no I/O operation outstanding, the dialog is aborted and the LINKMON process closes the server process. The server process should interpret the close operation as a dialog abort.

When writing requesters as process pairs with or without the TMF subsystem, note that it is not possible to checkpoint dialogs. The dialog and the transaction, if any, are aborted when the requester fails.

Cancellation of Server-Class Send Operations

The requester can explicitly cancel an outstanding server-class send operation if it has used the nowait option in the Pathsend procedure call and has not yet sensed completion of the server-class send operation through the AWAITIOX procedure. In this case, the server receives a Pathsend dialog abort system message (as described in Section 4) even if it has already replied to the last send operation.

Writing Requesters That Interoperate With NonStop TUXEDO Servers

Pathsend requesters can call on the services of NonStop TUXEDO servers in either of two ways:

- Directly, by making calls to the NonStop TUXEDO Application Transaction Monitor Interface (ATMI) functions
- Indirectly, by using the Pathway to TUXEDO (PWY2TUX) translation server

To communicate directly with a NonStop TUXEDO server by calling the ATMI functions, a Pathsend requester must also act as a NonStop TUXEDO client, as follows:

- It must be compiled as a NonStop Kernel Open System Services (OSS) process; therefore, it must be written in a Tandem language that supports Open System Services (such as C).
- It must link in the ATMI functions. For example, if the requester is a C program, it can be linked to the ATMI functions by being compiled with the `buildclient` command.
- It must join a NonStop TUXEDO application by calling the tpinit() ATMI function or one of the ATMI functions that implicitly calls tpinit(), such as tpalloc() or tpcall().
- It must follow the restrictions on Pathsend procedure calls that apply to NonStop TUXEDO clients: namely, it cannot make nowait calls (calls with bit 15 of the *`flags`* parameter set to 1) to the SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, and SERVERCLASS_DIALOG_SEND_ procedures.

A requester using this method of interoperation can use request/response messages, conversations, or both.

For more information about writing a NonStop TUXEDO client and calling the ATMI functions, refer to the *NonStop TUXEDO System Application Development Guide*. For the syntax of the ATMI function calls, refer to the *NonStop TUXEDO System Reference Manual*.

To communicate indirectly with a NonStop TUXEDO server by using the Pathway to TUXEDO translation server, a Pathsend requester program (or a SCREEN COBOL requester program) must be written according to the guidelines in the *NonStop TUXEDO System Pathway Translation Servers Manual*. Refer to that manual for further information. A Pathsend requester using the Pathway to TUXEDO translation server must use context-free Pathsend calls (calls to SERVERCLASS_SEND_); the translation server does not support dialogs (conversations).

4 Writing Pathway Servers

This section explains how to write server programs that service requests from Pathway requesters. Such requesters can be Pathsend requesters, SCREEN COBOL requesters, or clients that use the Remote Server Call (RSC) product or the Pathway Open Environment Toolkit (POET) product. Pathsend requesters are described in this manual. SCREEN COBOL requesters are described in the *Pathway/TS TCP and Terminal Programming Guide*; the SCREEN COBOL language is described in the *Pathway/TS SCREEN COBOL Reference Manual*. For information about writing RSC clients, refer to the *Remote Server Call (RSC) Programming Manual*. For information about writing POET clients, refer to the *Pathway Open Environment Toolkit (POET) Programming Manual*.

Pathway server programs read requests from $RECEIVE, as described in the *Guardian Programmer's Guide* and in the manuals describing Tandem programming languages. You can write Pathway server programs in C, C++, COBOL85, Pascal, pTAL, TAL, FORTRAN, or Extended BASIC. You should be familiar with the Guardian requester/server model and with the $RECEIVE mechanism as implemented in the programming language you are using.

Note. This section describes how to write Pathway servers in the Guardian environment. You can also write a Pathway server program in the NonStop Kernel Open System Services (OSS) environment. The basic design considerations in this section apply also to Pathway servers in the Open System Services environment; however, additional Open System Services programming considerations also apply. For information about these programming considerations, refer to the *Open System Services Programmer's Guide*.

If you are using the Transaction Management Facility (TMF) subsystem, you should also be familiar with general programming guidelines and considerations for TMF servers, as described in the *NonStop TM/MP Application Programmer's Guide*.

Basic Pathway Server Programming

The simplest type of Pathsend server is a context-free server. This subsection provides information related to writing context-free Pathway servers, as well as information that applies to all Pathway servers. Writing Context-Sensitive Servers, later in this section, provides information about the additional tasks required of a context-sensitive Pathway server.

In X/Open and NonStop TUXEDO system terminology, a context-free server is called a request/response server, and a context-sensitive server is called a conversational server.

Servers Shared by Different Types of Requesters

The protocol for Pathway server processes is essentially the same regardless of the type of requester they work with. Therefore, Pathway servers can be used by more than one type of requester; for example, by both Pathsend requesters and SCREEN COBOL requesters, or by both Guardian requesters and clients from client/server environments. If servers are used by several types of requester, the server program request and reply formats must be consistent with that of all the requesters.

Guardian Servers and Pathway Servers

Like a Guardian server, a Pathway server receives messages by reading the Guardian $RECEIVE file. However, unlike a Guardian server, it does not receive its messages directly from a requester program, but instead receives them from an intermediate process: either a LINKMON process or a terminal control process (TCP). Whereas a Guardian server receives open messages, I/O messages, and close messages from the requester, a Pathway server receives all these messages from the LINKMON process or the TCP. A Pathway server receives no information about the identity of the requester process that initiated the communication, unless the requester provides that information in the messages it sends.

The LINKMON process's open of the server process is shared among all the Pathsend requester processes running in the same processor as that LINKMON process. Therefore, a close does not necessarily occur when the communication with a particular requester process is finished.

Server Stop Protocol

A Pathway server must stop itself when the LINKMON process or the TCP closes it on behalf of the last requester that has a link to it. Otherwise, the PATHMON process considers the server process to be in the PENDING state.

Handling of Messages from $RECEIVE

COBOL85 servers always perform all of the I/O for the request message most recently read from $RECEIVE and always reply to that message before reading another message. For simplicity, this is the protocol servers should follow in most cases.

If you do need to write a multithreaded server or a context-sensitive server that performs queuing of messages from $RECEIVE, you must write this portion of your server program in a language other than COBOL85.

Pathsend Requester Failures

A Pathsend requester process can fail for reasons that include the following: the requester process calls ABEND, a processor fails, or someone stops the process.

When a Pathsend requester process fails with outstanding I/O operations, the effect is similar to that of a failure of any Guardian process with outstanding I/O; that is, all outstanding I/O operations are canceled, including those for outstanding server-class send operations. As a result, the target server process might get cancellation messages related to the outstanding calls to SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_.

However, the target server process does not get a close message, as it would have from a Guardian file-system open. Except in the case of a processor failure, the links that the LINKMON process established with server processes on behalf of the Pathsend requester are maintained, because these links are shared. Therefore, servers must monitor cancel messages, rather than close messages, to determine when a pending reply is no longer needed.

When a requester fails, all transactions initiated by that requester but not yet completed at the time of the failure are automatically aborted.

LINKMON Process Failures

If a halt occurs in the processor where a LINKMON process is running, all the Pathsend requesters in that processor fail. (See the previous subsection for details about requester failures.) The links that the LINKMON process established with the server processes are relinquished.

Linkage Space Considerations

If the server allocates too little space for $RECEIVE messages, system performance can be adversely affected, and the requester can get errors at peak workload times. To avoid this problem, the number of links specified by the server (for example, in the COBOL85 RECEIVE-TABLE OCCURS clause) should be greater than the value specified in the SET SERVER MAXLINKS parameter in PATHCOM. The server's specified number of links should be large enough for all openers, including requesters from other PATHMON environments that specify associative servers and from outside the Pathway environment (for example, operations applications).

The default value for MAXLINKS is an unlimited number of links; therefore, to avoid this problem, MAXLINKS must be set to a value. The error returned to a Pathsend requester is error 905 (FEScNoServerLinkAvailable) or 923 (FEScTooManyServerLinks). The error returned to a SCREEN COBOL requester is error 4 (link denied).

Considerations for Servers Used With SCREEN COBOL Requesters

SCREEN COBOL requesters require that the first two bytes of a server reply message contain an integer reply-code value. For more information about reply-code values, refer to the description of the REPLY CODE clause of the SCREEN COBOL SEND statement in the *Pathway/TS SCREEN COBOL Reference Manual*.

The checkpointing requirements of the Pathway/TS terminal control process (TCP) can be reduced significantly if TMF protected servers read outside of transaction mode—that is, perform their read operations while no transaction is in progress—before updating the database.

You can improve the performance of a server used by SCREEN COBOL requesters by taking advantage of the TCP's checkpointing strategy for TMF protected servers, as follows:

- Do not use transaction mode for a server with read-only access to a database if the requester displays the data before making any attempt to change it. In the event of a failure, the read operations are retryable and fault-tolerant operation is maintained.
- Do not use transaction mode for a server that writes to an entry-sequenced logging file in which duplicates are acceptable. In the event of a failure, the write operations can be rewritten.

For more information about the checkpointing strategy used by the TCP, refer to the *Pathway/TS TCP and Terminal Programming Guide*.

Consideration for Servers Used With Remote Server Call (RSC) Clients

Some Remote Server Call (RSC) clients are written to check for an integer reply-code value in the first two bytes of the server's reply message. If the RSC client program calls the RscSetOption function to set TMF_OPTION to either 2 (RSC_END_TRANS) or 3 (RSC_BEGIN_END_TRANS), the RSC transaction delivery process (TDP) monitors this reply code from the server to determine whether the client should end the transaction. The reply code is a feature that can allow optimization to reduce network traffic. For further information about use of the RSC reply code, refer to the *Remote Server Call (RSC) Programming Manual*.

Nested Servers

A Pathway server can use Pathsend procedure calls to make requests to servers in other server classes. In such a case, the server is acting as a server with respect to the requester that sends requests to it, but it is also acting as a Pathsend requester with respect to another server. Servers communicating with each other in this way are known as nested servers. Because they are Pathsend programs, nested servers must be written in C, C++, COBOL85, Pascal, pTAL, or TAL.

If you use nested servers, you should try to prevent queues of requests from developing, because a single-threaded server that calls Pathsend procedures waits for a response before proceeding. Servers usually just wait for disk I/O, a high-priority activity. Waiting for a low-priority server might tie up system resources.

For information about how to code the requester functions of nested servers (that is, how to use the Pathsend procedure calls), refer to Writing Pathsend Requesters. An example of a nested server, called PATHSRV, is given in Example B-2 on page B-53.

Using Context-Free Servers With Context-Sensitive Requesters

Dialogs using context-sensitive requesters are described under Context-Sensitive Pathsend Programming in Section 3. A context-free server-class send operation (initiated by a call to SERVERCLASS_SEND_) is, in effect, a dialog consisting of a single send operation. Pathway servers that are coded to be context-free can participate in single-send dialogs with context-sensitive requesters. However, if these servers check for system messages, you must modify them so that they recognize Pathsend dialog abort system messages, as described under Writing Context-Sensitive Servers later in this section.

Considerations for Servers That Use the TMF Subsystem

If you are writing a Pathway server that uses the TMF subsystem for transaction management, a number of additional programming considerations apply, related to the following topics:

- Recommended application structure
- Writing a server to use the TMF subsystem if your application does not follow the recommended structure
- Using audited and nonaudited files
- Record locking
- Grouping transaction operations

Recommended Structure for Applications

The recommended structure for applications that use the TMF subsystem is described in this subsection. If your current or planned application has these characteristics, programming the application to use the TMF subsystem is relatively straightforward; otherwise, refer to Writing a Server to Use the TMF Subsystem, later in this section, for a checklist of changes to make.

One process (usually the requester) coordinates all of the work required to do a single TMF transaction; this process identifies the beginning and ending points of each transaction. Additionally, if the server replies to a request message by indicating that it failed to complete all of the changes, this process can either abort and abandon the transaction or abort and retry the transaction.

The communication between requesters and servers is by standard interprocess I/O. The requester does the send operation, and the server does the READUPDATE call for $RECEIVE and the REPLY call. Each request message and the server's reply to the message is for a single transaction.

Any disk I/O request is for a single transaction. The TMF subsystem appends the process's current transaction identifier to each disk-request message so that the audit trails can include the identity of the transaction responsible for each database change.

How concurrency control is performed depends on which relational database management system (RDBMS) is being used. If the NonStop SQL/MP relational database management system is used, concurrency control is done by means of NonStop SQL/MP access options. Use of the access options causes NonStop SQL/MP software to generate the appropriate TMF transactions as required; therefore, servers that use NonStop SQL/MP databases are not required to include TMF procedure calls or statements. For information about use of the NonStop SQL/MP access options, refer to the *NonStop SQL/MP Reference Manual*. For servers that use the Enscribe database record manager, concurrency control is done by using the Enscribe locking facilities, and you must program the transactions by using TMF procedure calls or statements. For

information about use of the Enscribe locking facilities, refer to the *Enscribe Programmer's Guide*.

Servers do not reply to request messages until all work for the request has been completed. The contents of the reply message indicate the outcome of the request, which is one of the following:

- All the work for the request was completed successfully.
- None of the work for the request was completed.
- The work for the request was only partially completed.

In the first case, the requester can commit the transaction. In the second case, the requester can commit the transaction and then retry it. In both of these cases, the information in the server's reply is sufficient to ensure the integrity of the transaction.

However, if the transaction work was only partially completed, as in the third case, the server needs to ensure that the transaction is not committed so that the incomplete work can be backed out. To ensure transaction backout, the server should call the ABORTTRANSACTION procedure after reading the server's request and before sending its reply. A call to ABORTTRANSACTION by the server does not end the transaction—only the requester can end it—but such a call imposes the requirement that the requester also call ABORTTRANSACTION, rather than ENDTRANSACTION, after the requester's reply.

Writing a Server to Use the TMF Subsystem

If, for some reason, your application does not follow the structure described in Recommended Structure for Applications, earlier in this section, you should consult the checklist that follows.

Writing a Pathway server to use the TMF subsystem requires that you do the following:

1. Decide which files in the database should be audited.
2. Determine any modifications that are necessary to convert the application to follow the TMF subsystem locking rules.
3. Decide how to group sequences of the application operations into TMF transactions (that is, units of recovery).
4. Ensure that any fault-tolerant servers respond correctly to file-system error 75: REQUESTING PROCESS HAS NO CURRENT-TRANSACTION IDENTIFIER. A backup server that takes over in mid-transaction does not have a current-transaction identifier to send to the disk process; therefore, the disk process returns error 75 to the server, which passes the error to the requester. If the requester aborts and retries the transaction, the new request has a current-transaction identifier. The preferred solution is to change the servers so they are not fault-tolerant servers.
5. Determine whether any new transaction deadlock situations are introduced as a result of TMF implicit record locking and modify the application to avoid the deadlock. (Transaction deadlock is a situation in which two transactions cannot

proceed because they are each waiting for the other to release a lock.) One way to cope with deadlock is to use timeout.

In addition, your requester program must use the necessary transaction-control procedure calls or statements to begin and end the transaction and to abort or restart the transaction if necessary.

Note. Whenever your server begins work on a new queued message, it must call the ACTIVATERECEIVETRANSID procedure to change the current transaction identifier, as described in the *NonStop TM/MP Application Programmer's Guide*.

Using Audited and Nonaudited Files

TMF recovery strategy involves backing out the aborted transaction changes; backing out those changes enables the transaction to be reexecuted from the beginning (with a new transaction identifier). This strategy means that if you decide to have a mixture of audited and nonaudited files in the database, you must be careful: only changes to audited files are backed out. If a transaction works on a mix of audited and nonaudited files, the operations on the nonaudited files must be retryable.

A retryable operation is an operation that can be interrupted and repeated an indefinite number of times without affecting the consistency of the database; for example, all reading operations are retryable. Whether or not a writing operation (on a nonaudited file) is retryable depends on your criteria for consistency of the data in the database. If the transaction changes both audited and nonaudited files, you should analyze the transaction to determine whether backing it out and reexecuting it affects consistency.

For example, consider a transaction that extracts records from a database, computes some aggregates like averages or means, and then uses the aggregates to extract a subset of the extracted records from the database for summary reporting. This transaction can be implemented by doing the extraction twice, the first time to compute the aggregates and the second time to extract the subset. You can place the extracted records in a nonaudited scratch file (each server can have its own scratch file, to avoid conflict among them). If the transaction is aborted and restarted, the transaction starts writing the scratch file from the beginning and there is no need for the scratch file to be audited.

Another example is logging all input messages to a server, which allows examination of them after a failure. It is self-defeating to designate the log file as an audited file; the message that caused the failure would be backed out.

The following restrictions apply to applications that use the TMF subsystem:

- When working with audited files, do not use the COBOL85 feature that provides record blocking on an unstructured disk file.
- When changing audited files, do not use a server that provides its own record blocking, its own record caching, or its own form of record locking.

Locking Records

If your application uses the TMF subsystem, your servers must follow the TMF locking rules. Locking gives the TMF subsystem the control required to ensure that transactions are presented with a consistent view of the database. With respect to the locking of records, you must consider the following aspects of your application:

- Repeatable reads.
- Errors that result from locks being held by the transaction identifier instead of the process identifier and OPENID of the file opener.
- Errors that result from reading deleted records.
- Batch updates by a transaction that acquires a large number of locks. You should use file locks instead of record locks for batch updating.

For details about how to handle these aspects of your application, refer to the *NonStop TM/MP Application Programmer's Guide*.

Grouping Transaction Operations

Your application can view the transaction as a logical unit of work; for example, the order header and all of the detail items in a purchase order might be such a work unit. The TMF subsystem, however, treats the transaction as a physical unit of recovery. When you use the TMF subsystem in your application, you must consider this difference.

Basically, you need to answer certain questions. What is the logical unit of work that you want to accomplish within an application? How can the work be divided into a number of transactions that can be recovered by the TMF subsystem?

Factors that influence the answers to these questions are:

- Concurrency: How long will record locks be held by a transaction?
- Performance: How much server activity and how many display screens are involved in the choice of one conversion strategy over another?
- Consistency: Are the units of recovery large enough to ensure that your criteria for consistency will be maintained?

In view of these factors, two guidelines can help you decide how to group the database accesses made by an application into a single transaction:

- Any group of accesses that together modify the database from one consistent state to another consistent state should be a single TMF transaction.
- Any group of accesses that require a consistent view of the database should be a single TMF transaction.

The following examples demonstrate how you might apply these guidelines.

Example 1

Some logical transactions do not have to be identified as TMF transactions. For example, a logical transaction locates a single record and displays the record contents. Because this transaction changes nothing in the database, it does not affect consistency and does not have to be a TMF transaction.

Example 2

A data-entry transaction with a group of accesses that insert new data into the database should be a TMF transaction. For example, a logical transaction records receipt of some items for a stockroom by accepting the stock codes and quantity received from a data-entry operator and then updates the records (in an audited file) for the items.

Because the first guideline applies, you should arrange to begin a TMF transaction after the data is accepted and to end the transaction after the last record is updated. The TMF subsystem ensures that all changes resulting from the one operator entry either are permanent or are backed out in case the transaction aborts. Note that because any change to an audited file requires a transaction identifier, this example is also true if the transaction inserts only one record in the file.

Example 3

An update transaction should be a TMF transaction. For example, assume a logical transaction does the following:

1. Accepts a specification from the operator
2. Performs the equivalent of an inquiry operation to find the data that will be updated
3. Releases the locks obtained for the inquiry
4. Displays the data for the operator
5. Accepts modifications to the displayed data (saving a copy of the original displayed data)
6. Performs the inquiry a second time
7. Verifies that the results of the first inquiry and the second inquiry are the same
8. Writes the modified record to the database

The transaction should be implemented as two TMF transactions. The first should begin after the data is accepted and should end (rather than release the locks) after the last record is read. The second should begin after the modifications to the displayed data have been accepted and should end after the last modified record is written to the database. If the inquiry part of the transaction is just a single read, however, there is no need for the first inquiry to be part of a TMF transaction.

Servers as Process Pairs

When you are using the TMF subsystem, you should not write your server processes as fault-tolerant process pairs. The additional programming and functional overhead required to do so is unnecessary. If your servers are already coded as process pairs, however, it is not usually necessary to change them back to ordinary servers. For such servers, note that if the primary server process fails, the backup process (on takeover) does not have a current-transaction identifier. This means that the server process receives error 75: NO CURRENT-TRANSACTION IDENTIFIER on the first I/O request to an audited file. You should insert code into such a server either to recognize this error and report it as a failure to the requester, or to terminate when it receives this error.

Because the COBOL85 run-time library recognizes the PARAM named NONSTOP, you can prevent a COBOL85 server from running as a process pair by having a PARAM NONSTOP OFF in effect when the server is started. For Pathway servers, you can accomplish this task by including PARAM NONSTOP OFF with the parameters in the definition of the server class during PATHMON configuration. When PARAM NONSTOP OFF is in effect when the server is started, the COBOL85 run-time library ignores the STARTBACKUP and CHECKPOINT verbs and stores the successful completion code in the PROGRAM-STATUS special register.

There is no comparable mechanism to indicate that a server process coded in TAL, pTAL, or FORTRAN should not run as a process pair. If you have server processes in these languages that are coded as process pairs, you can either implement a custom PARAM or recode your server.

Transaction Deadlocks

An application that uses the TMF subsystem might hold more record locks and hold them longer than it would without the TMF subsystem because:

- Implicit locks are held on the keys of deleted records.
- Implicit locks are held for inserted records.
- Locks are held until the transaction is either committed or aborted and backed out.

The increased locking could cause new possibilities for transaction deadlock. If transaction deadlock might become a problem, consider implementing the methods for coping with deadlock discussed in the *Guardian Programmer's Guide*.

Considerations for Debugging Pathway Servers

When you are running a Pathway server in debug mode (that is, with the server class configured with DEBUG ON) or when the server falls into debug mode because of an error, some situations might cause errors to be returned to the requester communicating with that server, and in some cases also might cause errors to be returned to other requesters. Those situations are described in the following subsections.

LINKMON Process and TCP Timeouts

The LINKMON process and the TCP have a built-in five-minute timeout for process opens and I/O operations such as get-link requests. During normal operation, such operations are generally completed in less than five minutes unless there is a problem. However, if the server process is in debug mode, the server can be stopped (for example, at a breakpoint) for longer than five minutes. In certain cases, this can cause a process open or I/O operation from a LINKMON process or TCP to time out. The effects of the timeout depend on what type of operation timed out.

If an I/O operation to the PATHMON process times out, the LINKMON process or the TCP behaves as if there were a problem with the PATHMON process: that is, it shuts down communication with the PATHMON process, relinquishing all links granted by that PATHMON process.

In this situation, if the requester that sent the request that timed out was a Pathsend requester, this request (as well as all other server-class send requests queued under that PATHMON process) fails with Pathsend error 902, FEScPathmonConnect, with file-system error 40 (timeout error). Subsequent send requests to server classes under that PATHMON process fail with Pathsend error 915, FEScPathmonShutDown.

If the timed-out request came from a SCREEN COBOL requester, this request (as well as all other server-class send requests queued under that PATHMON process) fails with a SEND error with a TERMINATION-STATUS value of 18 (PATHMON I/O error) and a TERMINATION-SUBSTATUS value of 40. Subsequent send requests to server classes under that PATHMON process fail with a SEND error with a TERMINATION-STATUS value of 18 (PATHMON I/O error) and a TERMINATION-SUBSTATUS value of 40.

If a timeout occurs on the server process open, the LINKMON process or the TCP returns the link to the PATHMON process. If other links to the server class exist, no send requests fail as a result of the open failure. If no other links exist, the send request (as well as all other server-class send requests queued for that LINKMON process or TCP) fails. A Pathsend requester receives Pathsend error 904, FEScServerLinkConnect, with file-system error 40 (timeout error). A SCREEN COBOL requester receives a SEND error with a TERMINATION-STATUS value of 12 (I/O error) and a TERMINATION-SUBSTATUS value of 40 (timeout error).

Note also that if the server class is configured so that it can be opened more than once (that is, with MAXLINKS greater than 1), a LINKMON process or TCP can attempt to open a server process in that server class at any time. This open attempt can time out if the server process is being debugged.

When Pathsend error 915 or a SEND error with TERMINATION-STATUS 12 occurs with a timeout error during server debugging, use the PATHCOM STATUS PATHMON command to find the server classes that are in the LOCKED state. Identify the server program file for each locked server class, and issue the TACL command STATUS *, PROG `object-file-name` to list all running processes. Stop these processes by using the TACL STOP command. The PATHMON process then unlocks the server class, the LINKMON process or the TCP completes the shutdown logic, and error 915 is no longer returned.

The LINKMON process now has no more communication with that PATHMON process. When a subsequent send request comes in for a server controlled by that PATHMON process, the LINKMON process or TCP opens the PATHMON process, and waits to be opened by that PATHMON process in turn, before initiating any get-link requests to that PATHMON process.

PATHMON Process Timeouts

Similarly, the PATHMON process has a built-in five-minute timeout for I/O operations, such as the opening of a a server process and the sending of a startup message to it that can occur as a result of a get-link request. If such an operation takes longer than five minutes (for example, because the server process is in debug mode), the get-link request from the LINKMON process or the TCP times out. A Pathsend requester receives Pathsend error 902, FEScPathmonConnect, with file-system error 40 (timeout error). A SCREEN COBOL requester receives a SEND error with a TERMINATION-STATUS value of 18 (PATHMON I/O error) and a TERMINATION-SUBSTATUS value of 40 (timeout error).

By default, the COBOL85 and C run-time libraries handle initialization automatically before the first line of server code is executed. Therefore, PATHMON timeout errors on server initialization are most likely to occur when you are debugging servers written in TAL or pTAL, or other servers that are explicitly programmed to monitor system messages such as open messages.

Server Timeouts

Another kind of timeout problem that can occur for servers being debugged is related to the configured timeout value for the server class (set in the PATHCOM SET SERVER TIMEOUT command). This timeout covers only the time taken by the I/O to the server process. If a timeout occurs, the LINKMON process or the TCP cancels the send operation, and the Pathsend call or SCREEN COBOL SEND request fails. Changing the value of the timeout parameter during debugging might help prevent this problem from occurring.

Avoiding Timeout Errors

A good way to avoid most timeout errors during the debugging of servers is to write an Inspect command file that automatically continues server operation before a timeout occurs.

Writing Context-Sensitive Servers

If you are writing a context-sensitive Pathway server, you must follow the guidelines in Basic Pathway Server Programming earlier in this section and also perform additional programming tasks. This subsection describes these additional tasks and other considerations for programming context-sensitive servers.

When you use context-sensitive Pathway servers, the requester and server must be designed to work together. The context-sensitive Pathsend procedure calls used by the requester, described in Section 3, Writing Pathsend Requesters, and Section 5, Pathsend Procedure Call Reference, convey dialog information to the server, and the server conveys dialog information in its reply.

Note. The SCREEN COBOL language does not support context sensitivity; therefore, to take advantage of context sensitivity, you must use either a Pathsend requester or a GDSX front-end process that uses the GDSX pseudo Pathsend procedures.

Functions of a Context-Sensitive Server

In addition to the functions performed by all Pathway servers, a context-sensitive server must do the following:

- Detect a newly established dialog
- Receive, service, and reply to messages associated with a dialog
- Correlate messages with a dialog
- Continue a dialog
- Terminate a dialog
- Abort a dialog
- Detect an aborted dialog

It is simpler to code a context-sensitive server if you allow only one dialog at a time. To impose this restriction, you must configure the server class with a MAXLINKS value of 1. If MAXLINKS is set to a value other than 1, you must code your server to save multiple dialog contexts and to switch context, if needed, on each incoming request.

When a server receives a message on $RECEIVE, it checks dialog flag bits <12:13> returned by the file-system procedure FILE_GETRECEIVEINFO_ or the Common Run-Time Environment (CRE) procedure CRE_Receive_Read_ to determine whether this is the first message of a new dialog or a message within an existing dialog. A value of zero in these two bits indicates a context-free send operation. In addition, the server can check dialog flag bit 14 to determine the model used by the requester for associating transactions with dialogs.

For the syntax of the FILE_GETRECEIVEINFO_ procedure, refer to the *Guardian Procedure Calls Reference Manual*. For the syntax of the CRE_Receive_Read_ procedure, refer to the *Common Run-Time Environment (CRE) Programmer's Guide*.

Note. CRE_Receive_Read_ and the other CRE routines are callable only from TAL, pTAL, FORTRAN, and COBOL85 programs and are used when you are writing mixed-language programs. You cannot use CRE routines along with Guardian procedure calls. For more information about the use of the CRE routines, refer to the *Common Run-Time Environment (CRE) Programmer's Guide*.

Detecting a Newly Established Dialog

A context-sensitive server detects a newly established dialog by checking bits <12:13> of the dialog flags returned by the FILE_GETRECEIVEINFO_ or the CRE_Receive_Read_ procedure. If the value in these bits is 1, the message is the first in a new dialog. This message corresponds to the requester's call to the SERVERCLASS_DIALOG_BEGIN_ procedure; this call starts the dialog and usually also sends data.

Receiving, Servicing, and Replying to Messages in a Dialog

The server receives, services, and replies to messages in a dialog by doing the following:

1. Reads a message from $RECEIVE by using the $RECEIVE reading mechanism for the programming language, such as a call to the Guardian READUPDATE or READUPDATEX procedure.
2. Checks bits <12:13> of the dialog flags returned by the FILE_GETRECEIVEINFO_ or the CRE_Receive_Read_ procedure. If the value in these bits is 1, the message is the first in a new dialog. If the value is 2, the message is for an existing dialog.
3. Checks the value of bit 14 of the dialog flags if the TMF subsystem is being used, the server is enforcing transaction commit protection during the dialog, and the value of bits <12:13> of the dialog flags is 1 (first message in a new dialog). If the value of bit 14 is 1, aborts the dialog by replying with FEEOF (1).
4. Processes the message and performs the requested services.
5. Issues a reply by using the language's $RECEIVE writing mechanism, such as a call to the REPLY or REPLYX procedure.

Dialog Control

The server controls the dialog by means of a file-system error code it returns in its reply. Table 4-1 shows the effect of the reply error codes.

Table 4-1. Meaning of Error Codes Returned by Context-Sensitive Server in Reply

Error Returned by Server	Meaning
70 (FEContinue)	Allow the dialog to continue.
0 (FEOK)	End the dialog. After the server replies with this code, the associated TMF transaction (if any) will be allowed to commit, and the server will receive no further sends associated with this dialog.
1 (FEEOF)	Abort the dialog. After the server replies with this code, there will be no further sends associated with this dialog.
Any other value[1]	Abort the dialog. After the server replies with this code, there will be no further sends associated with this dialog. The requester receives this error as the file-system error of the FEScServerLinkConnect error.

[1]Use of other error values is not recommended because of their effect on the link to the server. Refer to accompanying text for details.

Use of error values other than 1 (FEEOF) is not recommended. Any other error value causes the LINKMON process to close the link to the server and return it to the PATHMON process. If there are no other links to that server process, the server process is deleted; then, if that process is later needed to service subsequent requests, it will need to be re-created, thereby affecting system performance. It is recommended, therefore, that servers always reply with FEEOF to abort a dialog. The server should return any additional information in the message itself rather than in the reply code.

Whether the current TMF transaction, if any, is also aborted when an error value is returned depends on the setting of dialog flag bit 14 in the call to SERVERCLASS_DIALOG_BEGIN_, as explained in the following subsection.

Use of TMF Transactions With Dialogs

A context-sensitive server participates in a TMF transaction in the same manner as context-free servers would do. The server inherits the transaction identifier of the received message. The server can abort the transaction when it has the transaction identifier.

When the server calls FILE_GETRECEIVEINFO_ or CRE_Receive_Read_ and detects the first message in a dialog, it can also check the dialog flags used to initiate the dialog. If you want to enforce transaction commit protection during the dialog—that is, ensure that the requester cannot call ENDTRANSACTION until the server has ended the dialog—you can code your server to verify that bit 14 of the dialog flags is 0 whenever the value of bits <12:13> is 1. If the requester's value for bit 14 is 1, the server can abort the dialog by replying to the message with an error value of FEEOF (1).

When the one-transaction-per-dialog model is used (when bit 14 of the dialog flags is equal to 0), the server must end the dialog by replying FEOK to a message before the requester can commit the associated TMF transaction. To allow requesters to ask the server to end the dialog, you can program your server to recognize a user-defined "end request" command.

When the one-transaction-per-dialog model is used, the transaction is automatically aborted when the server returns an error value (FEEOF, or any other value besides FEOK or FEContinue) in the reply. When the any-transaction-per-dialog model is used (when bit 14 of the dialog flags is equal to 1), no automatic transaction abort occurs. In either case, the requester should abort the transaction when it receives an error reply.

Correlating Messages With a Dialog

After a message with a "dialog begin" indication has been received, all messages received on that open of $RECEIVE will pertain to that dialog until one of the following occurs:

- The server terminates the dialog.
- A Pathsend dialog abort system message (system message -121) is received, indicating that the dialog has been aborted. To receive Pathsend dialog abort messages, the server must be monitoring system messages of this type. For more information about Pathsend dialog abort messages, refer to "Detecting an Aborted Dialog" later in this section.
- A path error (such as CPU down or network down) occurs on the link to the LINKMON process.

Continuing a Dialog

To indicate that the dialog should continue, the server replies to a message in the dialog with an error value of FECONTINUE (70).

Aborting a Dialog

The server can abort a dialog by replying to any message in the dialog with an error value of FEEOF (1).

Terminating a Dialog

After the dialog has started, either the requester or the server can abort it, but only the server can end it. To end the dialog, the server replies to a message in the dialog with an error value of FEOK (0).

Detecting an Aborted Dialog

When a dialog is aborted, either explicitly or by termination of the requester, the server receives a Pathsend dialog abort system message (system message -121), if the server is monitoring system messages of this type. Context-sensitive Pathsend servers must ensure that Pathsend dialog abort system messages are monitored on $RECEIVE. The Pathsend dialog abort system message is described in the *Guardian Procedure Errors and Messages Manual*.

When a server receives a Pathsend dialog abort system message, the server should call FILE_GETRECEIVEINFO_ to obtain the file number and process handle associated with the message. The server can then use these parameters to identify the dialog. The server should reply to the Pathsend dialog abort system message with an error value of

either FEOK (0) or FEEOF (1); these values direct the LINKMON process to release the link for re-use.

If a processor or network failure occurs, it is possible to have a dialog abort but not receive a Pathsend dialog abort message. Therefore, to monitor all aborted dialogs, your server must also monitor CPU down, remote CPU down, and loss of communication with network node system messages if it uses FILE_GETRECEIVEINFO_, or CLOSE system messages if it uses CRE_Receive_Read_ or the COBOL85 or FORTRAN language. The server should treat such messages as dialog aborts.

Servers can monitor CPU down, remote CPU down, and loss of communication with network node system messages by calling the Guardian procedures MONITORCPUS and MONITORNET, which are described in the *Guardian Programmer's Guide*.

It is also possible to get Pathsend dialog abort messages that do not represent actual aborted dialogs. A server can receive such a message if the requester calls SERVERCLASS_DIALOG_BEGIN_ and then cancels the message. In that case, the LINKMON process sends the Pathsend dialog abort message even though the server might not have received the first message in the dialog. Similarly, if the requester calls SERVERCLASS_SEND_ with the nowait option and then cancels the operation before being notified of completion of the operation through the AWAITIOX procedure, the server receives the Pathsend dialog abort message even if it has already replied to the last send operation.

A cancel operation can also occur if the requester abends while a server-class send operation is in progress, whether or not the send operation was invoked with the nowait option.

Writing Pathway Servers That Interoperate With TUXEDO Requesters

A Pathway server can service requests from TUXEDO requesters (TUXEDO clients or TUXEDO servers acting as clients). Such requesters can make their requests to a Pathway server in either of two ways:

- Directly, by making calls to the Pathsend procedures described in this manual
- Indirectly, by using the TUXEDO to Pathway (TUX2PWY) translation server

Only those TUXEDO requesters that are NonStop TUXEDO native System /T clients can make direct requests by using Pathsend procedure calls. For more information, refer to the *NonStop TUXEDO System Application Development Guide*.

Native System /T clients, remote TUXEDO requesters using System /Domain, and non-native TUXEDO workstation clients can indirectly invoke the services of a Pathway server by using the TUXEDO to Pathway translation server. If you use this translation server, special guidelines apply, and you might need to modify the code of existing Pathway servers. For further information, refer to the *NonStop TUXEDO System Pathway Translation Servers Manual*.

5 Pathsend Procedure Call Reference

Pathsend programs use six procedures that are part of the Guardian procedure library. This section provides the syntax and semantics of these procedure calls, preceded by information about how to call the procedures from each of the languages supported by Pathsend. The descriptions are in alphabetic order by procedure name.

For each procedure, the reference information includes:

- A brief description of what the procedure does
- The syntax of the procedure call (in TAL)
- Parameter definitions
- Additional usage considerations for the call, where applicable

Lengthy usage considerations that apply to several different Pathsend procedures are grouped at the end of the section.

You invoke all of the procedures by using a function-type statement in which the procedure returns a numeric completion code (as an integer value).

Table 5-1 lists the Pathsend procedure calls by name and gives the purpose of each one.

Table 5-1. Summary of Pathsend Procedure Calls

Procedure	Purpose
SERVERCLASS_DIALOG_ABORT_	Requests that the specified dialog be aborted
SERVERCLASS_DIALOG_BEGIN_	Initiates a dialog with a server process in a server class and sends the first message in the dialog
SERVERCLASS_DIALOG_END_	Ends the specified dialog
SERVERCLASS_DIALOG_SEND_	Initiates a send within the specified dialog
SERVERCLASS_SEND_	Initiates a context-free send operation to a server process in the specified server class
SERVERCLASS_SEND_INFO_	Gets error information about the last call to any of the other Pathsend procedures

The procedures whose names begin with SERVERCLASS_DIALOG_ are used for requests to context-sensitive servers. The SERVERCLASS_SEND_ procedure is used for requests to context-free servers. The SERVERCLASS_SEND_INFO_ procedure is used to obtain information about both context-free and context-sensitive requests.

The method for accessing these procedures from a Pathsend program depends on the programming language you use. You can write Pathsend programs in C, C++,

COBOL85, Pascal, pTAL, or TAL. The topics that follow explain how to call the Pathsend procedures from each of the supported programming languages.

Note. For general information about calling these and other Tandem system procedures from programs written in various programming languages, refer to the information about accessing Guardian procedures in the *Guardian Programmer's Guide*.

None of the Pathsend procedures set the condition-code register. Therefore, language restrictions on procedures that set this register do not apply to the Pathsend procedures.

Note. For some Pathsend procedures, some parameters in the TAL calling syntax contain an embedded colon and are of the form *name*:*length*. These parameters are TAL parameter pairs. For further information about parameter pairs and their use in mixed-language programming (for example, calls to these procedures from languages other than TAL or pTAL), refer to the *TAL Programmer's Guide*.

Calls From C or C++

To invoke any of the procedures from within a C or C++ program, you execute a statement of the following form:

```
error = SERVERCLASS_SEND_ ( pathmon-process-name
                            ,pathmon-process-name-len
                            ,server-class-name
                            ,server-class-name-len
                            ,message-buffer
                            ,request-len
                            ,maximum-reply-len
                            ,actual-reply-len
                            ,timeout
                            ,flags
                            ,scsend-op-num
                            ,tag );
```

error

is an integer variable defined earlier in your data declarations.

pathmon-process-name, *pathmon-process-name-len*, *server-class-name*, *server-class-name-len*, *message-buffer*, *request-len*, *maximum-reply-len*, *actual-reply-len*, *timeout*, *flags*, *scsend-op-num*, and *tag*

are variables defined earlier in your data declarations. The types of these variables should be the C types that correspond to the TAL variable types specified in the Pathsend procedure-call description later in this section. For a table of these corresponding data types, refer to the information about mixed-language programming in the *C/C++ Programmer's Guide*.

To use the Pathsend procedures in a C program, you must first have named them in an #include <cextdecs> preprocessor directive.

For further information, refer to the *C/C++ Programmer's Guide*.

Calls From COBOL85

To invoke any of the procedures from within a COBOL85 program, you execute a statement of the following form:

```
ENTER "SERVERCLASS_SEND_" USING pathmon-process-name
                                pathmon-process-name-len
                                server-class-name
                                server-class-name-len
                                message-buffer
                                request-len
                                maximum-reply-len
                                actual-reply-len
                                timeout
                                flags
                                scsend-op-num
                                tag
                          GIVING error
```

pathmon-process-name, *pathmon-process-name-len*, *server-class-name*, *server-class-name-len*, *message-buffer*, *request-len*, *maximum-reply-len*, *actual-reply-len*, *timeout*, *flags*, *scsend-op-num*, and *tag*

are variables defined in the WORKING-STORAGE SECTION of the DATA DIVISION. The types of these variables should be the COBOL85 types that correspond to the TAL variable types specified in the Pathsend procedure-call description later in this section. For a table of these corresponding data types, refer to the information about invoking non-COBOL routines in the *COBOL85 Manual*.

If the length of a string parameter is declared in a separate parameter (as in SERVER_CLASS_SEND_), this parameter must be passed to the procedure. If the length is declared as part of the string parameter in the form *name*:*length* (as in SERVERCLASS_DIALOG_BEGIN_ and SERVERCLASS_DIALOG_SEND_), the length must not be passed explicitly.

error

is an integer variable (USAGE NATIVE-2) defined in the WORKING-STORAGE SECTION of the DATA DIVISION.

For further information, refer to the *COBOL85 Manual*.

Calls From Pascal

To invoke any of the procedures from within a Pascal program, you execute a statement of the following form:

```
error := SERVERCLASS_SEND_ ( pathmon-process-name
                            ,pathmon-process-name-len
                            ,server-class-name
                            ,server-class-name-len
                            ,message-buffer
                            ,request-len
                            ,maximum-reply-len
                            ,actual-reply-len
                            ,timeout
                            ,flags
                            ,scsend-op-num
                            ,tag )
```

error

is a variable of type IO_Error_Number defined earlier in your data declarations.

pathmon-process-name, *pathmon-process-name-len*, *server-class-name*, *server-class-name-len*, *message-buffer*, *request-len*, *maximum-reply-len*, *actual-reply-len*, *timeout*, *flags*, *scsend-op-num*, and *tag*

are variables defined earlier in your data declarations. The types of these variables should be the Pascal types that correspond to the TAL variable types specified in the Pathsend procedure-call description later in this section. For definitions of these corresponding data types, refer to the information about mixed-language programming and data-type correspondence in the *Pascal Reference Manual*.

To use the Pathsend procedures in a Pascal program, you must first have named them in a SOURCE PEXTDECS compiler directive. For further information, refer to the *Pascal Reference Manual*.

Calls From TAL or pTAL

To invoke any of the procedures from within a TAL or pTAL program, you execute a statement of the following form:

```
error := SERVERCLASS_SEND_ ( pathmon-process-name
                             ,pathmon-process-name-len
                             ,server-class-name
                             ,server-class-name-len
                             ,message-buffer
                             ,request-len
                             ,maximum-reply-len
                             ,actual-reply-len
                             ,timeout
                             ,flags
                             ,scsend-op-num
                             ,tag );
```

error

is an integer variable defined earlier in your data declarations.

pathmon-process-name, *pathmon-process-name-len*, *server-class-name*, *server-class-name-len*, *message-buffer*, *request-len*, *maximum-reply-len*, *actual-reply-len*, *timeout*, *flags*, *scsend-op-num*, and *tag*

are variables defined earlier in your data declarations, with types as specified in the Pathsend procedure-call description later in this section.

To use the Pathsend procedures in a TAL or pTAL program, you must first have named them in a SOURCE EXTDECS compiler directive. For further information, refer to the *TAL Programmer's Guide*.

SERVERCLASS_DIALOG_ABORT_ Procedure

The SERVERCLASS_DIALOG_ABORT_ procedure aborts the specified dialog.

A call to SERVERCLASS_DIALOG_BEGIN_ to begin a dialog must be matched by a call to SERVERCLASS_DIALOG_ABORT_ or SERVERCLASS_DIALOG_END_ at the end of the dialog.

Syntax

The syntax of the SERVERCLASS_DIALOG_ABORT_ procedure is:

```
error := SERVERCLASS_DIALOG_ABORT_ ( dialog-id );          ! i
```

error returned value

INT

returns an error word containing one of the following values:

0 (FEOK) indicates that the call was successful.

233 (FESCErr) indicates that an error occurred. You can call the SERVERCLASS_SEND_INFO_ procedure to get more detailed information about the error.

dialog-id input

INT(32):value

is the dialog identifier previously returned from the SERVERCLASS_DIALOG_BEGIN_ call that began the dialog.

This parameter is required.

Considerations

The following considerations apply to the SERVERCLASS_DIALOG_ABORT_ procedure:

- If the server has opened $RECEIVE for system message handling and is using the Common Run-Time Environment (CRE), aborting the dialog will cause the server to receive a system message -121, PATHSEND DIALOG ABORT.
- SERVERCLASS_DIALOG_SEND_ operations in progress within this dialog at the time of the call to SERVERCLASS_DIALOG_ABORT_ are canceled.

SERVERCLASS_DIALOG_BEGIN_ Procedure

The SERVERCLASS_DIALOG_BEGIN_ procedure initiates a dialog with a server process in a server class and sends the first message in the dialog.

The procedure identifies the server class to the system and returns a dialog identifier for subsequent dialog operations. A SERVERCLASS_DIALOG_BEGIN_ call must be matched by a SERVERCLASS_DIALOG_ABORT_ or SERVERCLASS_DIALOG_END_ call at the end of the dialog.

The completion of this processing—that is, getting the final outcome (success or failure) and, if successful, the reply data—occurs in one of two ways, depending on whether the send operation is initiated as waited or nowait:

- For waited send operations, initiation and completion are both performed by the SERVERCLASS_DIALOG_BEGIN_ procedure.
- For nowait send operations, initiation is performed by the SERVERCLASS_DIALOG_BEGIN_ procedure, and completion is performed by calling the AWAITIOX procedure.

Syntax

The syntax of the SERVERCLASS_DIALOG_BEGIN_ procedure is:

```
error := SERVERCLASS_DIALOG_BEGIN_
                         ( dialog-id                        ! o
                          ,pathmon-process-name:length      !
i:i
                          ,server-class-name:length         !
i:i
                          ,message-buffer                   !
i,o
                          ,request-len                      ! i
                          ,maximum-reply-len                ! i
                          ,[ actual-reply-len ]             ! o
                          ,[ timeout ]                      ! i
                          ,[ flags ]                        ! i
                          ,[ scsend-op-num ]                ! o
                          ,[ tag ] );                       ! i
```

error returned value

INT

returns an error word containing one of the following values:

0 (FEOK) indicates that the call was successful and the server has ended the dialog.

70 (FEContinue) indicates that the call was successful and the server is ready for the next message in the dialog.

233 (FESCErr) indicates that an error occurred. You can call the SERVERCLASS_SEND_INFO_ procedure to get more detailed information about the error.

dialog-id output

INT:ref:EXT(32):1

returns an identifier that can be used for subsequent operations on this dialog.

This parameter is required.

pathmon-process-name:*length* input:input

STRING:ref:EXT:*, INT:value

contains the external Guardian process name of the PATHMON process controlling the server class (for example, $PM or \AB.$PMN). The process name portion can have up to five characters after the dollar sign ($) if it is a local process name and up to four characters after the dollar sign if it is a process on a remote system. The name cannot include an optional first or second name qualifier, must be left justified in the buffer, and can contain trailing blanks.

The integer value is the byte length of the *pathmon-process-name* string. This value can range from 2 through 15.

This parameter is required.

server-class-name:*length* input:input

STRING:ref:EXT:*, INT:value

contains the name of the server class to send to (for example, EMP-SERVER). This name must conform to the Pathway server-class naming rules, must be left justified in the buffer, and can contain trailing blanks. This server-class name, along with the *pathmon-process-name*, uniquely identifies a server class.

The integer value is the byte length of the *server-class-name* string. This value can range from 1 through 15.

This parameter is required.

message-buffer input, output

STRING:ref:EXT:*

contains the message to send to the server class. On successful completion of the send operation, *message-buffer* contains the reply from the server class.

This parameter is required.

request-len input

INT:value

is the byte length of the data contained in *message-buffer*. The range of acceptable values is 0 through 32767 bytes.

This parameter is required.

maximum-reply-len input

INT:value

is the maximum number of bytes that the reply message from the server class can contain. The range of acceptable values is 0 through 32767 bytes.

No more than *maximum-reply-len* bytes of the actual reply are placed into *message-buffer* upon successful completion of a send.

It is not an error if the server replies with a byte count not equal to the *maximum-reply-len* value specified by the requester in the call to this procedure. If the server replies with a byte count greater than the *maximum-reply-len* value, the actual bytes transferred are truncated to *maximum-reply-len.*

This parameter is required.

actual-reply-len output

INT:ref:EXT:1

returns a count of the number of bytes returned in the server process reply. This parameter is for waited I/O only and can be omitted for nowait I/O. The return value of this parameter is 0 if nowait I/O is used. For nowait I/O, the actual reply length is returned by AWAITIOX.

timeout input

INT(32):value

specifies the maximum amount of time, in hundredths of a second, that the LINKMON process waits for the completion of this send. This value must be either -1D or a value greater than 0D. The default is -1D (wait indefinitely).

If there is an outstanding I/O operation to a server process when a SERVERCLASS_DIALOG_BEGIN_ operation times out, the I/O operation is canceled.

See Timeout Considerations for Pathsend Programming later in this section for details about timeout for waited and nowait operations.

flags input

INT:value

flags.<15>

with a value of 1 indicates that this operation is to be performed nowait. A value of 0 indicates that this operation is to be performed waited. The default is 0.

flags.<14>

if set to 0, selects the one-transaction-per-dialog model. In this model, the dialog records the transaction identifier that is current at the time of the SERVERCLASS_DIALOG_BEGIN_ call. Subsequent SERVERCLASS_DIALOG_SEND_, SERVERCLASS_DIALOG_ABORT_, and SERVERCLASS_DIALOG_END_ calls that use the returned *dialog-id* must specify this transaction identifier, or the calls will fail. ENDTRANSACTION will fail unless the dialog has been ended (not aborted).

When this bit is set to 0, the TMF subsystem treats a dialog like an I/O operation: the ENDTRANSACTION operation fails until the dialog has finished. The same restriction applies to a nested server (a server that receives a request and then becomes a requester to other servers): if a server receives a message in a dialog and then initiates a dialog with another server, it must complete the entire initiated dialog before replying to the message from the received dialog.

A value of 1 selects the any-transaction-per-dialog model. In this model, all server-class send operations within the dialog will contain the transaction identifier that is current at the time of the send, and there are no restrictions on ENDTRANSACTION other than those associated with calls to the WRITEREAD procedure.

The default is 0.

flags.<0:13>

must be 0.

scsend-op-num output

INT:ref:EXT:1

returns the server-class send operation number. You can use the server-class send operation number in place of the file-number parameter in calls to CANCEL, CANCELREQ, and AWAITIOX for nowait sends, and in calls to FILEINFO for waited and nowait sends, to indicate that the calls refer to server-class send operations. The value of *scsend-op-num* is determined on the first successfully initiated nowait send. This value is returned on every subsequent nowait send that is initiated successfully. A value of -1 is returned for nowait sends that are not initiated successfully. A value of -1 is always returned for waited sends.

See Server-Class Send Operation Number later in this section for more information about the server-class send operation number.

tag input

INT(32):value

is used for nowait I/O only. The *tag* is stored by the system and then passed back to the application by the AWAITIOX procedure when the nowait operation is completed. You can use the *tag* parameter to identify multiple nowait I/O operations. For waited I/O, this parameter is not used and can be omitted. The default is 0D.

Considerations

If the SERVERCLASS_DIALOG_BEGIN_ procedure fails but does not return a valid dialog identifier, the dialog was never created. In this case, there is no need to abort the dialog.

For additional considerations, refer to Usage Considerations for Pathsend Procedures at the end of this section.

SERVERCLASS_DIALOG_END_ Procedure

The SERVERCLASS_DIALOG_END_ procedure cleans up resources for the specified dialog after the server has ended it.

A call to SERVERCLASS_DIALOG_BEGIN_ must be matched by a call to SERVERCLASS_DIALOG_ABORT_ or SERVERCLASS_DIALOG_END_ at the end of the dialog. For the SERVERCLASS_DIALOG_END_ procedure to work correctly, the server must previously have ended the dialog by replying with an *error* value other than FEContinue (70).

This procedure does not perform any I/O operations.

Syntax

The syntax of the SERVERCLASS_DIALOG_END_ procedure is:

```
error := SERVERCLASS_DIALOG_END_ ( dialog-id );          ! i
```

error returned value

INT

returns an error word containing one of the following values:

0 (FEOK) indicates that the call was successful.

233 (FESCErr) indicates that an error occurred. You can call the SERVERCLASS_SEND_INFO_ procedure to get more detailed information about the error.

dialog-id input

INT(32):value

is the dialog identifier previously returned from the SERVERCLASS_DIALOG_BEGIN_ call that began the dialog.

This parameter is required.

Considerations

None.

SERVERCLASS_DIALOG_SEND_ Procedure

The SERVERCLASS_DIALOG_SEND_ procedure initiates a send within the specified dialog.

The completion of this processing—that is, getting the final outcome (success or failure) and, if successful, the reply data—occurs in one of two ways, depending on whether the send operation is initiated as waited or nowait:

- For waited send operations, initiation and completion are both performed by the SERVERCLASS_DIALOG_SEND_ procedure.
- For nowait send operations, initiation is performed by the SERVERCLASS_DIALOG_SEND_ procedure, and completion is performed by the AWAITIOX procedure.

This procedure is similar to the context-free SERVERCLASS_SEND_ procedure, with a few differences as described under "Considerations."

Syntax

The syntax of the SERVERCLASS_DIALOG_SEND_ procedure is:

```
error := SERVERCLASS_DIALOG_SEND_
                            ( dialog-id                          ! i
                             ,message-buffer                     !
i,o
                             ,request-len                        ! i
                             ,maximum-reply-len                  ! i
                             ,[ actual-reply-len ]               ! o
                             ,[ timeout ]                        ! i
                             ,[ flags ]                          ! i
                             ,[ scsend-op-num ]                  ! o
                             ,[ tag ] );                         ! i
```

error returned value

INT

returns an error word containing one of the following values:

0 (FEOK) indicates that the call was successful and the server has ended the dialog.

70 (FEContinue) indicates that the call was successful and the server is ready for the next message in the dialog.

233 (FESCErr) indicates that an error occurred. You can call the SERVERCLASS_SEND_INFO_ procedure to get more detailed information about the error.

dialog-id `input`

INT(32):value

is an identifier, previously returned from SERVERCLASS_DIALOG_BEGIN_, that specifies the dialog for this send operation.

This parameter is required.

message-buffer `input, output`

STRING:ref:EXT:*

contains the message to send to the server class. On successful completion of the send operation, *message-buffer* contains the reply from the server class.

This parameter is required.

request-len `input`

INT:value

is the byte length of the data contained in *message-buffer*. The range of acceptable values is 0 through 32767 bytes.

This parameter is required.

maximum-reply-len `input`

INT:value

is the maximum number of bytes that the reply message from the server class can contain. The range of acceptable values is 0 through 32767 bytes.

No more than *maximum-reply-len* bytes of the actual reply are placed into *message-buffer* upon successful completion of a send.

It is not an error if the server replies with a byte count not equal to the *maximum-reply-len* value specified by the requester in the call to this procedure. If the server replies with a byte count greater than the *maximum-reply-len* value, the actual bytes transferred are truncated to *maximum-reply-len.*

This parameter is required.

actual-reply-len `output`

INT:ref:EXT:1

returns a count of the number of bytes returned in the server process reply. This parameter is for waited I/O only and can be omitted for nowait I/O. The return value of this parameter is 0 if nowait I/O is used. For nowait I/O, the actual reply length is returned by AWAITIOX.

timeout input

INT(32):value

specifies the maximum amount of time, in hundredths of a second, that the LINKMON process waits for the completion of this send. This value must be either -1D or a value greater than 0D. The default is -1D (wait indefinitely).

If there is an outstanding I/O operation to a server process when the SERVERCLASS_DIALOG_SEND_ operation times out, the I/O operation is canceled.

See Timeout Considerations for Pathsend Programming later in this section for details about timeout for waited and nowait operations.

flags input

INT:value

flags.<15>

with a value of 1 indicates that this operation is to be performed nowait. A value of 0 indicates that this operation is to be performed waited. The default is 0.

flags.<0:14>

must be 0.

scsend-op-num output

INT:ref:EXT:1

returns the server-class send operation number. You can use the server-class send operation number in place of the file-number parameter in calls to CANCEL, CANCELREQ, and AWAITIOX for nowait sends, and in calls to FILEINFO for waited and nowait sends, to indicate that the calls refer to server-class send operations. The value of *scsend-op-num* is determined on the first successfully initiated nowait send. This value is returned on every subsequent nowait send that is initiated successfully. A value of -1 is returned for nowait sends that are not initiated successfully. A value of -1 is always returned for waited sends.

See Server-Class Send Operation Number later in this section for more information about the server-class send operation number.

tag input

INT(32):value

is used for nowait I/O only. The *tag* is stored by the system and then passed back to the application by the AWAITIOX procedure when the nowait operation is completed. You can use the *tag* parameter to identify multiple nowait I/O operations. For waited I/O, this parameter is not used and can be omitted.
The default is 0D.

Considerations

The SERVERCLASS_DIALOG_SEND_ procedure is similar to the context-free SERVERCLASS_SEND_ procedure, with the following differences:

- The dialog identifier, obtained from the SERVERCLASS_DIALOG_BEGIN_ call that started the dialog, is used to identify the dialog, which is associated with a particular server class.
- The SERVERCLASS_DIALOG_SEND_ call fails if the current transaction identifier does not match the transaction identifier used for the SERVERCLASS_DIALOG_BEGIN_ call, unless this feature has been overridden by setting the *flags*.<14> bit to 1 in the call to SERVERCLASS_DIALOG_BEGIN_.
- The send fails if there is already an outstanding send in the dialog from a previous nowait call to this procedure.

For additional considerations, refer to Usage Considerations for Pathsend Procedures at the end of this section.

SERVERCLASS_SEND_ Procedure

The SERVERCLASS_SEND_ procedure initiates a context-free send operation to a server process in the specified server class.

The completion of this processing—that is, getting the final outcome (success or failure) and, if successful, the reply data—occurs in one of two ways, depending on whether the send operation is initiated as waited or nowait:

- For waited send operations, initiation and completion are both performed by the SERVERCLASS_SEND_ procedure.
- For nowait send operations, initiation is performed by the SERVERCLASS_SEND_ procedure, and completion is performed by calling the AWAITIOX procedure.

Syntax

The syntax of the SERVERCLASS_SEND_ procedure is:

```
error := SERVERCLASS_SEND_ ( pathmon-process-name        ! i
                            ,pathmon-process-name-len    ! i
                            ,server-class-name           ! i
                            ,server-class-name-len       ! i
                            ,message-buffer              ! i,o
                            ,request-len                 ! i
                            ,maximum-reply-len           ! i
                            ,[ actual-reply-len ]        ! o
                            ,[ timeout ]                 ! i
                            ,[ flags ]                   ! i
                            ,[ scsend-op-num ]           ! o
                            ,[ tag ] );                  ! i
```

error returned value

INT

returns an error word containing one of the following values:

0 (FEOK) indicates that the call was successful.

233 (FESCErr) indicates that an error occurred. You can call the SERVERCLASS_SEND_INFO_ procedure to get more detailed information about the error.

pathmon-process-name `input`

STRING:ref:EXT:*

contains the external Guardian process name of the PATHMON process controlling the server class (for example, $PM or \AB.$PMN). The process name portion can have up to five characters after the dollar sign ($) if it is a local process name and up to four characters after the dollar sign if it is a process on a remote system. The name cannot include an optional first or second name qualifier, must be left justified in the buffer, and can contain trailing blanks.

This parameter is required.

pathmon-process-name-len `input`

INT:value

is the byte length of the *pathmon-process-name* string. This value can range from 2 through 15.

This parameter is required.

server-class-name `input`

STRING:ref:EXT:*

contains the name of the server class to send to (for example, EMP-SERVER). This name must conform to the Pathway server-class naming rules, must be left justified in the buffer, and can contain trailing blanks. This server-class name, along with the *pathmon-process-name*, uniquely identifies a server class.

This is a required parameter.

server-class-name-len `input`

INT:value

is the byte length of the *server-class-name* string. This value can range from 1 through 15.

This parameter is required.

message-buffer `input, output`

STRING:ref:EXT:*

contains the message to send to the server class. On successful completion of the send operation, *message-buffer* contains the reply from the server class.

This parameter is required.

request-len input

INT:value

is the byte length of the data contained in *message-buffer*. The range of acceptable values is 0 through 32767 bytes.

This parameter is required.

maximum-reply-len input

INT:value

is the maximum number of bytes that the reply message from the server class can contain. The range of acceptable values is 0 through 32767 bytes.

No more than *maximum-reply-len* bytes of the actual reply are placed into *message-buffer* upon successful completion of a send.

It is not an error if the server replies with a byte count not equal to the *maximum-reply-len* value specified by the requester in the call to this procedure. If the server replies with a byte count greater than the *maximum-reply-len* value, the actual bytes transferred are truncated to *maximum-reply-len.*

This parameter is required.

actual-reply-len output

INT:ref:EXT:1

returns a count of the number of bytes returned in the server process reply. This parameter is for waited I/O only and can be omitted for nowait I/O. The return value of this parameter is 0 if nowait I/O is used. For nowait I/O, the actual reply length is returned by AWAITIOX.

timeout input

INT(32):value

specifies the maximum amount of time, in hundredths of a second, that the LINKMON process waits for the completion of this send. This value must be either -1D or a value greater than 0D. The default is -1D (wait indefinitely).

If there is an outstanding I/O operation to a server process when a SERVERCLASS_SEND_ operation times out, the I/O operation is canceled.

See Timeout Considerations for Pathsend Programming later in this section for details about timeout for waited and nowait operations.

flags input

INT:value

flags.<15>

with a value of 1 indicates that this operation is to be performed nowait. A value of 0 indicates that this operation is to be performed waited. The default is 0.

flags.<0:14>

must be 0.

scsend-op-num output

INT:ref:EXT:1

returns the server-class send operation number. You can use the server-class send operation number in place of the file number parameter in calls to CANCEL, CANCELREQ, and AWAITIOX for nowait sends, and in calls to FILEINFO for waited and nowait sends, to indicate that the calls refer to server-class send operations. The value of *scsend-op-num* is determined on the first successfully initiated nowait send. This value is returned on every subsequent nowait send that is initiated successfully. A value of -1 is returned for nowait sends that are not initiated successfully. A value of -1 is always returned for waited sends.

See Server-Class Send Operation Number later in this section for more information about the server-class send operation number.

tag input

INT(32):value

is used for nowait I/O only. The *tag* is stored by the system and then passed back to the application by the AWAITIOX procedure when the nowait operation is completed. You can use the *tag* parameter to identify multiple nowait I/O operations. For waited I/O, this parameter is not used and can be omitted. The default is 0D.

Considerations

Refer to Usage Considerations for Pathsend Procedures at the end of this section.

SERVERCLASS_SEND_INFO_ Procedure

The SERVERCLASS_SEND_INFO_ procedure retrieves error information about the last SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, SERVERCLASS_DIALOG_SEND_, SERVERCLASS_DIALOG_END_, or SERVERCLASS_DIALOG_ABORT_ operation that was initiated or completed with return error 233 (FEScError) or 0 (FEOK).

If the return error from the previous Pathsend call is 0 (FEOK), both the Pathsend error and the file-system error will always be 0, so you do not need to call SERVERCLASS_SEND_INFO_.

Syntax

The syntax of the SERVERCLASS_SEND_INFO_ procedure is:

```
error := SERVERCLASS_SEND_INFO_ ( pathsend-error        ! o
                                 ,file-system-error );  ! o
```

error returned value

INT

returns an error word. This error is associated with the call to SERVERCLASS_SEND_INFO_ and not with the previous Pathsend call. The error word contains one of the following file-system errors:

0 (FEOK) indicates that no errors occurred in the call to SERVERCLASS_SEND_INFO_.

2 (FEInvalOp) is returned if the caller has an invalid segment in use. Error 2 is also returned if the caller has no extended data segment in use and one of the reference parameters is an extended address.

22 (FEBoundsErr) is returned if a reference parameter is out of bounds.

29 (FEMissParam) is returned if a required parameter is missing.

These errors are programming errors.

pathsend-error output

INT:ref:1

returns the Pathsend error. See Section 6, Pathsend Errors, for descriptions of Pathsend errors.

file-system-error output

INT:ref:1

returns the file-system error. See the *Guardian Procedure Errors and Messages Manual* for descriptions of file-system errors.

Considerations

The following considerations apply to the SERVERCLASS_SEND_INFO_ procedure:

- The condition code setting has no meaning following a call to SERVERCLASS_SEND_INFO_.
- A call to SERVERCLASS_SEND_INFO_ before a call is ever made to SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ results in return error 0 (FEOK), Pathsend error 906 (FEScNoSendEverCalled), and file-system error 0 (FEOK).

Usage Considerations for Pathsend Procedures

The following subsections discuss usage considerations that apply to several of the Pathsend procedure calls.

Condition Code

The condition-code setting has no meaning following a Pathsend procedure call.

Waited I/O

The following considerations apply to waited Pathsend procedure calls:

- The *tag* parameter has no meaning and can be omitted.
- On a successful completion of a waited Pathsend procedure call, the *actual-reply-len* parameter indicates the number of bytes in the reply.

For an example of issuing a waited call to SERVERCLASS_SEND_, see the Pathsend server program example PATHSRV, Example B-2 on page B-53. In this COBOL85 program, paragraph 460-SEND-TO-SUBSIDIARY-SERVER performs a waited SERVERCLASS_SEND_ call.

Nowait I/O

The following considerations apply to nowait Pathsend procedure calls:

- The maximum nowait depth for Pathsend procedure calls is 255 per process. In other words, a Pathsend requester process can have no more than 255 outstanding nowait server-class send operations at any one time.
- You complete a nowait Pathsend procedure call with a call to AWAITIOX. You cannot use the AWAITIO—without the X—procedure to complete a nowait server-class send operation. The SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, and SERVERCLASS_DIALOG_SEND_ procedures use a 32-bit extended message buffer address, and AWAITIO cannot be used to complete extended I/O operations.

  If a nowait Pathsend procedure call returns an error, the send operation was not initiated and therefore does not need to be completed with AWAITIOX.
- The *actual-reply-len* parameter has no meaning for nowait server-class send operations and can be omitted. The count of the number of bytes in the server-class reply is returned in the *count-transferred* parameter of the AWAITIOX procedure.
- After calling SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ with the nowait option, do not modify the I/O buffer before the I/O operation is completed as indicated by AWAITIOX.
- Nowait server-class send operations must not use buffers that are currently in use for other outstanding nowait I/O operations. The file system requires that these buffers not be modified.

- SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, and SERVERCLASS_DIALOG_SEND_ return the server-class send operation number in the `scsend-op-num` parameter. See Server-Class Send Operation Number later in this section for more information about the server-class send operation number.

For an example of issuing a nowait call to SERVERCLASS_SEND_, see the Pathsend program example BREQ, Example B-1 on page B-2. In this TAL program, the procedure Initiate^IO initiates a nowait SERVERCLASS_SEND_ call.

Calls Within a TMF Transaction

The following considerations apply when calling Pathsend procedures during a TMF transaction:

- If SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ is called during a TMF transaction, the transaction identifier is propagated to the server process in the same way as it is in all interprocess communication (for example, calls to the WRITEREAD procedure). As an application developer, you do not have to be concerned about the role of the LINKMON process in the propagation of transaction identifiers.
- For context-sensitive server-class send operations, two types of TMF protection are available, depending on the setting of bit 14 in the `flags` parameter on the call to SERVERCLASS_DIALOG_BEGIN_. For details, refer to the discussion of context-sensitive requesters in Section 3, Writing Pathsend Requesters.
- If a send is made to a server class that is configured with the TMF parameter set to OFF while there is a currently active transaction identifier, the send is completed with return error 233 (FEScError), Pathsend error 917 (FEScServerClassTmfViolation), and file-system error 0 (FEOK). See the *NonStop TS/MP System Management Manual* for details about the TMF parameter of the SET SERVER command.

Server-Class Send Operation Number

The server-class send operation number is returned in the `scsend-op-num` parameter on the first successfully initiated nowait server-class send operation (through a call to SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND). The same value is returned on every subsequent nowait send that is initiated successfully.

You can use the server-class send operation number in the following calls:

- In calls to AWAITIOX to wait for completion of any SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ operation
- In calls to CANCEL and CANCELREQ to cancel an outstanding SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ operation
- In calls to FILEINFO

A value of -1 is returned for nowait send operations that are not initiated successfully. A value of -1 is always returned for waited send operations.

Note. Passing the server-class send operation number as a `filenum` parameter to Guardian procedures other than AWAITIOX, CANCEL, CANCELREQ, and FILEINFO results in file-system error 2 (FEInvalOp).

Calling AWAITIOX

You can use the server-class send operation number as the file number in calls to the AWAITIOX procedure to wait for completion of any outstanding SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ operation. Alternatively, you can specify -1 as the file number to AWAITIOX to wait for completion of any outstanding I/O operation, including calls to SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_. When completion of one of these calls occurs, you can identify the specific call by the tag returned by AWAITIOX. (You cannot use the AWAITIO—without the X—procedure to complete a nowait server-class send operation. The SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, and SERVERCLASS_DIALOG_SEND_ procedures use a 32-bit extended message buffer address, and AWAITIO cannot be used to complete extended I/O operations.)

For an example of one way to use AWAITIOX, see the Pathsend program example BREQ, Example B-1 on page B-2. In this TAL example, the procedure complete^io waits for completion of an I/O operation on any file and identifies the I/O operation by the tag returned by AWAITIOX. You can also see how this procedure calls AWAITIOX. If AWAITIOX returns with an error (condition code CCL), it calls FILEINFO to retrieve the error code. If the error returned is 233 (FEScError), SERVERCLASS_SEND_INFO_ is called to get the specific Pathsend error and file-system error.

Canceling a Server-Class Send Operation

You cancel an outstanding SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ operation by using the Guardian CANCEL or CANCELREQ procedure if you have used the nowait option in the Pathsend procedure call and have not yet received notification of completion of the server-class send operation with the AWAITIOX procedure. Note that if the AWAITIOIX call was completed with a Pathsend error, such as a server-class send timeout error (described in the following subsection), this is still a completion; in such cases, you should not call CANCEL or CANCELREQ.

Calling CANCEL or CANCELREQ has the following effects on your Pathsend program:

- The program is not affected by any reply to the SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ call.
- The program cannot determine whether the request message has been sent to a server process, and if it was sent, whether or not it was canceled before the server process finished processing the request.

- If the canceled call was to SERVERCLASS_DIALOG_BEGIN_ or SERVERCLASS_DIALOG_SEND_, the dialog is aborted and the server receives a Pathsend dialog abort system message, even if the server has already replied to the server-class send operation.
- If the SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ operation was performed within a TMF transaction, the transaction is automatically aborted.

On a call to CANCELREQ, you supply:

- The server-class send operation number as the file-number parameter.
- The tag identifying the specific operation to be canceled. (The tag is optional.)

If you use the tag parameter, the system cancels the oldest incomplete SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ call with that tag value. If you do not provide a tag, the system cancels the oldest incomplete SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ call.

On a call to CANCEL, you supply the server-class send operation number as the file-number parameter.

Refer to the *Guardian Procedure Calls Reference Manual* for descriptions of the CANCEL and CANCELREQ procedures.

A cancel operation can also occur if the requester abends while a server-class send operation is in progress, whether or not the send operation was invoked with the nowait option.

Calling FILEINFO

You can use the server-class send operation number returned by SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ as the file-number parameter in calls to FILEINFO to get the return error associated with the last waited or nowait SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ call. FILEINFO returns the same error that the server-class procedure call returned. See the `error` parameter in the procedure-call syntax descriptions earlier in this section for details about the return errors.

Timeout Considerations for Pathsend Programming

When you design a Pathsend application, you can decide which of two timeout methods that you want the LINKMON process to use:

- A server TIMEOUT attribute that applies only to the server process I/O
- A SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ parameter value that applies to the entire processing of the send request

Because the errors for each are different, programs can differentiate between the two kinds of timeout.

Server Timeout

You can specify a TIMEOUT attribute for your Pathway servers. You specify the server TIMEOUT value when you configure the server. If a timeout occurs during an I/O operation to a server, the I/O operation is canceled and any TMF transaction is aborted. Unlike the timeout parameter to a SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ call, described in the following subsection, the TIMEOUT attribute does not include waiting for the server link; the TIMEOUT attribute applies only to the I/O to the server.

Server timeout returns Pathsend error 904 (FEScServerLinkConnect) and file-system error 40 (FETimedOut).

Server-Class Send Timeout

Pathsend allows you to specify a timeout value on calls to SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, and SERVERCLASS_DIALOG_SEND_. You can specify a different timeout value for each call. For example, if you perform SERVERCLASS_SEND_ calls to local and remote systems, you can specify a shorter timeout value for the local sends and a longer value for the remote send operations.

Within a dialog, if `flags`.<14> was set to 0 in the call to SERVERCLASS_DIALOG_BEGIN_ and a timeout occurs, both the dialog and the transaction (if any) are automatically aborted.

If a waited SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ request is not completed before the specified timeout value expires, it is completed with return error 233 (FEScError), and a subsequent call to SERVERCLASS_SEND_INFO_ returns Pathsend error 918 (FeScSendOperationAborted) and file-system error 40 (FETimedOut). If there is an outstanding I/O operation to a server process when a SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ operation times out, that I/O operation is canceled and the transaction, if any, is automatically aborted.

Nowait server-class send operations are completed with a call to AWAITIOX. The timeout considerations for nowait operations are more complex, because a time-limit parameter can also be set in the AWAITIOX call. The error returned by AWAITIOX depends on which time limit was reached:

- If a timeout value is specified in a nowait call to SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ and the request is not completed within the specified time, the AWAITIOX call returns error 233, and a subsequent call to SERVERCLASS_SEND_INFO_ returns Pathsend error 918 (FeScSendOperationAborted) and file-system error 40 (FETimedOut).
- If a time-limit value is specified in the call to AWAITIOX and this time limit is reached, the AWAITIOX call returns file-system error 40 (FETimedOut).

If you use a time-limit value on AWAITIOX to wait on I/O operations other than the Pathsend calls SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, and SERVERCLASS_DIALOG_SEND_ (for example, WRITEREAD calls, which do not provide a timeout parameter) and you also use a timeout value on the Pathsend calls, timeouts can occur in one of the preceding two ways, depending on which timer expires first. To avoid this complex error handling, it is recommended that you use a timeout value only on the AWAITIOX call.

For programs whose only I/O operations are Pathsend procedure calls, it is not useful to use both a timeout on the Pathsend procedure calls and a time limit on AWAITIOX, because you would be timing the same operation twice. In this case, choose either Pathsend timeouts or AWAITIOX time limits.

When an AWAITIOX time limit is reached, whether an I/O operation is canceled depends on the `filenum` parameter used in the AWAITIOX call. If `filenum` is set equal to *`scsend-op-num`* and the AWAITIOX time-limit value is greater than zero, the oldest outstanding send operation is canceled. If *`filenum`* is set to-1 or if the AWAITIOX time-limit value is zero or less, no operation is canceled. When a Pathsend timeout is reached, no send operation is canceled regardless of parameter settings.

Note that any time a send is canceled, the current TMF transaction, if any, is automatically aborted. Any time a send timeout occurs when an I/O operation is outstanding on a server process, the I/O operation to the server is canceled and the transaction, if any, is aborted. However, you should code the requester to call the ABORTTRANSACTION procedure so that the appropriate cleanup is done on the requester’s side.

6 Pathsend Errors

This section describes the error codes that can be returned by the Pathsend procedure calls. These errors can be returned by processes that call the Pathsend procedures directly and also by software that uses these procedures internally, such as the NonStop TUXEDO system.

Types of Errors Returned by the Pathsend Procedures

Three errors are associated with each call to a Pathsend procedure (except SERVERCLASS_SEND_INFO_):

- A return error
- A Pathsend error
- A file-system error

The return error is the error returned by the Pathsend procedure or by AWAITIOX. For details about the return errors for a particular procedure call, see the `error` parameter in the syntax description for that call in Section 5, Pathsend Procedure Call Reference.

For waited send operations, the return error is returned from the Pathsend procedure call.

For nowait send operations, the return error is returned from the Pathsend procedure call if it is a send-initiation error. For errors other than send-initiation errors, the return error is returned by the AWAITIOX procedure and can be retrieved by calling the FILEINFO procedure.

If the return error is error 233 (FEScError), you can call SERVERCLASS_SEND_INFO_ to retrieve the Pathsend error and the file-system error. (If the return error is 0 (FEOK), both the Pathsend error and the file-system error are always 0, so you do not need to call SERVERCLASS_SEND_INFO_.)

The Pathsend errors are described in this section. The file-system errors are described in the *Guardian Procedures Errors and Messages Manual*; however, specific Pathsend considerations for some of the file-system errors are given in this section in the descriptions of the associated Pathsend errors.

Descriptions of Pathsend Errors

The Pathsend error codes are described in numeric order on the following pages. Each description includes the following:

- The error number and corresponding error literal
- The cause of the error
- Typical file-system errors that can be encountered with the Pathsend error
- The effect of the error

- How to recover from the error

These errors can be returned to Pathsend requesters. They can also be returned to NonStop TUXEDO clients that invoke NonStop TUXEDO request/response services or conversational services. When Pathsend errors are returned to a NonStop TUXEDO client, the necessary recovery actions might differ from those listed here; for information about recovery from errors in the NonStop TUXEDO environment, refer to the *NonStop TUXEDO System Messages Manual*.

For examples of how errors are returned to a Pathsend program, refer to the example programs in Appendix B, Examples. The error literals that follow the error numbers—for example, 233 (FEScError)—are used in the example programs.

You can recover from some errors by retrying the Pathsend procedure call; the errors you should retry depend on the requirements of your application. The programming examples in Appendix B, Examples, illustrate retrying the call for certain errors. You can use those examples and the error descriptions in this section to decide how to program for retrying errors.

900

```
FEScInvalidServerClassName
```

Cause. The server-class name specified in a call to SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ is not syntactically correct. This is a programming error.

Typical file-system error: 2 (FEInvalOp)

Effect. The send initiation fails with an error.

Recovery. Correct the server-class name syntax. See the *NonStop TS/MP System Management Manual* for a description of the correct syntax for server-class names.

901

```
FEScInvalidPathmonName
```

Cause. The PATHMON process name specified in a call to SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ is not syntactically correct. This name must be a valid external process name. This is a programming error.

Typical file-system error: 2 (FEInvalOp)

Effect. The send initiation fails with an error.

Recovery. Correct the PATHMON name syntax. Refer to Section 5, Pathsend Procedure Call Reference, for details about the correct syntax for PATHMON process names.

902

```
FEScPathmonConnect
```

Cause. An error has occurred in the requester's communication with the PATHMON process. For example, an open operation has failed, an I/O error has occurred, or the PATHMON process has failed.

Typical file-system errors: 12, 14, 40, 48, 201, or one of the path errors between 240 and 255.

Effect. The SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ call is completed with an error. The message is not sent to the server process.

Recovery. Recovery depends on the file-system error:

- Error 12 (FEInUse) indicates that the PATHMON process was unable to open the LINKMON process. Specific causes of this situation include (but are not limited to) the following:
  - The maximum number of LINKMON processes that the PATHMON process can communicate with has been exceeded. See the *NonStop TS/MP System Management Manual* for information about setting the MAXLINKMONS parameter.
  - The remote password on the system where the SERVERCLASS_SEND_ request originated was not set for the system where the server class was running. The PATHMON process receives a file-system error 48 but converts this error into a file-system error 12. To recover, ensure that all remote passwords are properly set, as described in the *Expand Network Management Guide*.
- Error 14 (FENoSuchDev) indicates the PATHMON process does not exist. Start the PATHMON process or use an existing PATHMON process.
- Error 40 (FETimeout) indicates that a timeout error occurred, possibly because a server was in debug mode. See Considerations for Debugging Pathway Servers in Section 4 for more information about timeout errors for servers in debug mode.
- Error 48 (FESecViol) indicates there was a security violation. See Section 3, Writing Pathsend Requesters, for information about network and server-class security.
- Error 201 (FEPathDown) or error 240 through 255 indicates that a path error occurred (for example, the PATHMON process failed). Restart the PATHMON process.

903

```
FEScPathmonMessage
```

Cause. The LINKMON process received an unrecognizable message from the PATHMON process while processing a SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ request. You might be using incompatible versions of the LINKMON and PATHMON processes, or this could be a LINKMON or PATHMON process internal error.

Typical file-system error: 0 (FEOK)

Effect. The SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ call is completed with an error. The message is not sent to the server process.

Recovery. This is a nonrecoverable error.

904

```
FEScServerLinkConnect
```

Cause. An error has occurred with the link to the server. For example, an open operation has failed or there is an I/O problem. This error could occur on a call to SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_.

Typical file-system errors: 14, 40, 48, 201, or one of the path errors between 240 and 255.

Effect. The call is completed with an error. The message might or might not have been sent to the server process, depending on the file-system error.

If the file-system error is a path error, any transaction associated with the call is aborted.

If this error is returned from SERVERCLASS_DIALOG_BEGIN_ or SERVERCLASS_DIALOG_SEND_, the dialog is ended on the server side.

Recovery. If this error is returned from SERVERCLASS_SEND_, recovery depends on the file-system error:

- Error 14 (FENoSuchDev) indicates that the server process does not exist. Retry the SERVERCLASS_SEND_ to cause the LINKMON process to use a different link to the server process or to receive additional links from the PATHMON process. The success of the retry depends on why the server process stopped.
- Error 40 (FETimedout) indicates that the I/O to the server process timed out because it exceeded the configured SERVER TIMEOUT value for the server class.
- Error 48 (FESecViol) indicates there was a security violation. Section 3, Writing Pathsend Requesters, for information about network and server-class security.

- Error 201 (FEPathDown) or error 240 through 255 indicates that a path error occurred (for example, the processor where the server process was running has failed).

If this error is returned from SERVERCLASS_DIALOG_BEGIN_ or SERVERCLASS_DIALOG_SEND_, use SERVERCLASS_DIALOG_END_ or SERVERCLASS_DIALOG_ABORT_ to terminate the requester's portion of the dialog.

905

```
FEScNoServerLinkAvailable
```

Cause. The LINKMON process had no links to the server class and was unable to get a link from the PATHMON process to satisfy this request. For example, MAXLINKS links for each server class are already allocated to other LINKMON processes and TCPs. This is a PATHMON configuration problem.

Typical file-system error: 0 (FEOK)

Effect. The SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ call is completed with an error. The message is not sent to the server process.

Recovery. Increase the maximum number of servers in the server class (with the PATHCOM MAXSERVERS parameter) or increase the number of links available to the server (with the PATHCOM MAXLINKS parameter). For details about the MAXSERVERS and MAXLINKS parameters, see the *NonStop TS/MP System Management Manual*.

906

```
FEScNoSendEverCalled
```

Cause. The SERVERCLASS_SEND_INFO_ procedure was called before SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ was ever called by this program. This is a programming error.

Typical file-system error: 0 (FEOK)

Effect. The SERVERCLASS_SEND_INFO_ call is completed with return error 0, Pathsend error 906, and file-system error 0.

Recovery. Ensure that your application program does not call SERVERCLASS_SEND_INFO_ before it calls SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_.

Note. Error 907, FEScInvalidSegmentId, and error 908, FEScNoSegmentInUse, are not returned on D-series systems. Error 908 has been replaced by error 912, FEScParameterBoundsError.

909

```
FEScInvalidFlagsValue
```

Cause. The caller set bits in the *flags* parameter that are reserved and must be 0. This is a programming error.

Typical file-system error: 2 (FEInvalOp)

Effect. The send initiation fails with an error.

Recovery. Set the reserved bits in the *flags* parameter to 0.

910

```
FEScMissingParameter
```

Cause. A required parameter was not supplied. This is a programming error.

Typical file-system error: 29 (FEMissParam)

Effect. The send initiation fails with an error.

Recovery. Check the syntax of the procedure call and supply the required parameters.

911

```
FEScInvalidBufferLength
```

Cause. The buffer length in the *request-len* or *maximum-reply-len* parameter for a call to SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ is invalid. This is a programming error.

Typical file-system error: 21 (FEBadCount)

Effect. The send initiation fails with an error.

Recovery. Check the buffer lengths allowed for the *request-len* and *maximum-reply-len* parameters in the syntax description for the procedure call, and specify the correct buffer lengths.

912

```
FEScParameterBoundsError
```

Cause. The address specified by a reference parameter is out of bounds, or the caller supplied a reference parameter that is an extended address but does not have an extended segment in use. This is a programming error.

Effect. The send initiation fails with an error.

Recovery. Correct the programming error.

913

```
FEScServerClassFrozen
```

Cause. The server class the process tried to send to is frozen.

Typical file-system error: 0 (FEOK)

Effect. The SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ call is completed with an error. The message is not sent to the server class.

Recovery. Resend after the system manager or operator has thawed the server class.

914

```
FEScUnknownServerClass
```

Cause. The server class is not configured through the specified PATHMON process. The program has specified an incorrect server-class name or specified the wrong PATHMON process.

Typical file-system error: 0 (FEOK)

Effect. The SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ call is completed with an error. The message is not sent to the server class.

Recovery. Check the server-class name and the PATHMON process name. Or check if the server class has been configured yet (the PATHMON-controlled objects could be in the process of being cold started).

915

```
FEScPathmonShutDown
```

Cause. The send operation has been denied for one of the following reasons:

- The PATHMON process for the server class is shutting down.
- A timeout occurred on an I/O operation to a server in debug mode.

Typical file-system error: 0 (FEOK)

Effect. The SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ call is completed with an error. The message is not sent to the server class.

Recovery. If the PATHMON process is shutting down, determine the reasons for the shutdown and perform appropriate recovery actions.

If a server process is in debug mode and a timeout error (file-system error 40) occurred, do the following:

- Use the PATHCOM STATUS PATHMON command to find the server classes that are in the LOCKED state.
- Identify the server program file for each locked server class.
- Issue the TACL command STATUS *, PROG `object-file-name` to list all running processes.
- Stop these processes by using the TACL STOP command.

For more information about timeout errors for servers in debug mode, refer to Considerations for Debugging Pathway Servers in Section 4.

916

```
FEScServerCreationFailure
```

Cause. The LINKMON process was unable to get a link to the server class due to a server creation failure. This is usually a server-class configuration problem, or a server might have a problem that causes it to fail.

Typical file-system error: 0 (FEOK)

Effect. The send initiation fails with an error.

Recovery. Verify that the server class has been configured correctly. If the problem is not in the configuration, correct the error in the server.

917

```
FEScServerClassTmfViolation
```

Cause. The transaction mode of the Pathsend program does not match that of the server class. The process has a current transaction ID at the time of the send, and the server class is configured with the TMF parameter set to OFF. This is a programming error or a server-class configuration error.

Typical file-system error: 0 (FEOK)

Effect. The call is completed with an error. The message is not sent to the server class.

Recovery. Correct your program or change the server-class configuration setting to TMF ON.

918

```
FEScSendOperationAborted
```

Cause. The send operation has been terminated at an indeterminate point.

Typical file-system error: 40 (FETimedOut)

Effect. The send fails. A message might or might not have been sent to the server process, depending on when the send was aborted.

Recovery. The recovery action depends on which file-system error has occurred. With error 40 (FETimedOut), you might want to try a larger timeout value.

919

```
FEScInvalidTimeoutValue
```

Cause. The caller supplied an invalid timeout value in a call to SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_. This is a programming error.

Typical file-system error: 2 (FEInvalOp)

Effect. The send initiation fails with an error.

Recovery. Specify a valid timeout value. See the procedure call syntax description for details about valid timeout values.

920

```
FEScPFSUseError
```

Cause. The caller's process file segment (PFS) could not be accessed.

Typical file-system error: 31 (FENoBufSpace)

Effect. The send initiation fails with an error.

Recovery. Code the process to stop itself if this error occurs.

921

```
FEScTooManyPathmons
```

Cause. A call to SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ specifies a PATHMON process not known to the LINKMON process, and the LINKMON process is already communicating with the maximum number of PATHMON processes allowed. The maximum number is 256.

Typical file-system error: 0 (FEOK)

Effect. The call is completed with an error. The message is not sent to the server process.

Recovery. In some cases, you can recover from this error by retrying the call. Whether a retry will work depends on the design and operating environment of your application. If the PATHMON processes in your application are frequently created and stopped, retry the call. Otherwise, investigate the cause of the large number of PATHMON processes and eliminate some processes.

For more information about LINKMON limits, refer to LINKMON Limit Errors on page 3-4 and also to the *NonStop TS/MP System Management Manual*.

922

```
FEScTooManyServerClasses
```

Cause. The LINKMON process already has links to the maximum number of server classes allowed for all PATHMON processes. The maximum number is 1024.

Typical file-system error: 0 (FEOK)

Effect. The SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ call is completed with an error. The message is not sent to the server process.

Recovery. In some cases, you can recover from this error by retrying the call. Whether a retry will work depends on the design and operating environment of your application.

For more information about LINKMON limits, refer to LINKMON Limit Errors on page 3-4 and also to the *NonStop TS/MP System Management Manual*.

923

```
FEScTooManyServerLinks
```

Cause. The LINKMON process already has the maximum number of concurrent links to server processes allowed for all PATHMON processes. The maximum number is 1750.

Typical file-system error: 0 (FEOK)

Effect. The SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ call is completed with an error. The message is not sent to the server process.

Recovery. In some cases, you can recover from this error by retrying the call. Whether a retry will work depends on the design and operating environment of your application.

For more information about LINKMON limits, refer to LINKMON Limit Errors on page 3-4 and also to the *NonStop TS/MP System Management Manual*.

924

```
FEScTooManySendRequests
```

Cause. The maximum number of concurrent server-class send operations allowed has been exceeded. The maximum number is 255 per requester and 512 for all requesters running in a processor. This error can occur on a call to SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_.

Typical file-system error: 0 (FEOK)

Effect. The call is completed with an error. The message is not sent to the server process.

Recovery. In some cases, you can recover from this error by retrying the call. Whether a retry will work depends on the design and operating environment of your application.

For more information about LINKMON limits, refer to LINKMON Limit Errors on page 3-4 and also to the *NonStop TS/MP System Management Manual*.

925

```
FEScTooManyRequesters
```

Cause. The LINKMON process is already communicating with the maximum number of requesters allowed. The maximum number of concurrently active Pathsend requesters per processor is 256. This error can occur only on the requester's first call to SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_.

Typical file-system error: 0 (FEOK)

Effect. The SERVERCLASS_SEND_ or SERVERCLASS_DIALOG_BEGIN_ call is completed with an error. The message is not sent to the server process.

Recovery. Retry if the number of Pathsend requesters fluctuates in this processor.

926

```
FEScDialogInvalid
```

Cause. The specified dialog identifier is not valid. This error can occur on a call to SERVERCLASS_DIALOG_SEND_, SERVERCLASS_DIALOG_END_, or SERVERCLASS_DIALOG_ABORT_.

Typical file-system error: 0 (FEOK)

Effect. The send initiation fails with an error.

Recovery. Use a valid dialog identifier.

927

```
FEScTooManyDialogs
```

Cause. The requester cannot start a new dialog, because it already has the maximum number of dialogs open. The maximum number of dialogs per requester is 256. This error can occur on a call to SERVERCLASS_DIALOG_BEGIN_.

Typical file-system error: 0 (FEOK)

Effect. The send initiation fails with an error.

Recovery. Reduce the number of dialogs.

928

```
FEScOutstandingSend
```

Cause. The requester has an outstanding send operation on this dialog. This error can occur on a call to SERVERCLASS_DIALOG_SEND_.

Typical file-system error: 0 (FEOK)

Effect. The send initiation fails with an error.

Recovery. Complete the current send before starting another.

929

```
FEScDialogAborted
```

Cause. The dialog has been aborted for one of the following reasons:

- The server requested a dialog abort.
- The server terminated between send operations.
- The server terminated immediately following send completion (with a reply of FEContinue), but before the LINKMON process had replied to the requester.

If the server terminated, the termination could be due to a server error or a network error. This error can occur on a call to SERVERCLASS_DIALOG_BEGIN_ or SERVERCLASS_DIALOG_SEND_.

Typical file-system error: if the server requested the abort, 1 (FEEOF); if the server terminated, the file-system error returned by the LINKMON process

Effect. The dialog is aborted. If *flags*.<14> was not set to 1 on the call to SERVERCLASS_DIALOG_BEGIN_, the transaction is also aborted. The procedure (if waited) or AWAITIOX (if nowait) returns with an error.

Recovery. Use SERVERCLASS_DIALOG_END_ to terminate the requester's portion of the dialog.

930

```
FEScChangedTransid
```

Cause. A call to SERVERCLASS_DIALOG_SEND_, SERVERCLASS_DIALOG_END_, or SERVERCLASS_DIALOG_ABORT_ was done under a different transaction identifier than the previous call to SERVERCLASS_DIALOG_BEGIN_ (which had specified *flags*.<14> = 0). This is a programming error.

Typical file-system error: 0 (FEOK)

Effect. The operation fails with an error.

Recovery. Use RESUMETRANSACTION to make the correct transaction identifier current, and reissue the call that failed.

931

```
FEScDialogEnded
```

Cause. A call to SERVERCLASS_DIALOG_SEND_ failed because the server had already ended the dialog, either by replying with FEOK or FEEOF or by terminating while a send was still outstanding.

Typical file-system error: the file-system error of the last server response. Any number other than 0 (FEOK) indicates one of the following:

- The dialog has been aborted.
- The previous call to SERVERCLASS_DIALOG_BEGIN_ or SERVERCLASS_DIALOG_SEND_ failed with an error that indicated an abort, but the requester has not yet aborted the dialog.

Effect. The procedure initiation fails with an error.

Recovery. None. Use SERVERCLASS_DIALOG_END_ to end the specified dialog or SERVERCLASS_DIALOG_ABORT_ to abort it.

933

```
FEScDialogOutstanding
```

Cause. A call to SERVERCLASS_DIALOG_END_ was made, but the server has not ended the dialog.

Typical file-system error: 0 (FEOK)

Effect. The procedure initiation fails with an error.

Recovery. Have the server end the dialog by replying FEOK, or use SERVERCLASS_DIALOG_ABORT_ to abort the dialog.

934

```
FEScTransactionAborted
```

Cause. The transaction associated with the dialog has been aborted.

Typical file-system error: 0 (FEOK)

Effect. The procedure initiation fails with an error. The dialog is now aborted.

Recovery. Use SERVERCLASS_DIALOG_END_ or SERVERCLASS_DIALOG_ABORT_ to terminate the requester's portion of the dialog.

947

```
FEScLinkmonConnect
```

Cause. There is a problem communicating with the LINKMON process in this processor.

Typical file-system errors: 14 (FENOSUCHDEV) and 43 (FENODISCSPACE)

Effect. The send initiation fails with an error.

Recovery. Recovery depends on the file-system error.

File-system error 14 indicates that there is no LINKMON process executing in the processor.

File-system error 43 indicates that the LINKMON process was unable to initialize itself; in this case, the LINKMON process writes a message to $0 that indicates the reason for the initialization failure. Each subsequent SERVERCLASS_SEND_, SERVERCLASS_DIALOG_BEGIN_, or SERVERCLASS_DIALOG_SEND_ call in this processor causes the LINKMON process to reattempt initialization; after the condition is corrected, the LINKMON process can complete initialization.

For information about initialization and limits for the LINKMON process, refer to LINKMON Limit Errors on page 3-4 and also to the *NonStop TS/MP System Management Manual*.

A NonStop TS/MP Limits for Pathsend Requesters

Table A-1 lists the NonStop TS/MP product limits that apply to the Pathsend programming environment.

For limits related to the configuration of a PATHMON environment, refer to the *NonStop TS/MP System Management Manual*.

Table A-1. Limits for Pathsend Requesters

Item	Limit
Buffer length	Maximum of 32,767 bytes per server-class send operation
Dialogs	Maximum of 256 per requester
	Maximum of 1750 for all requesters in a processor
	Per server, as many as the number of allowed links. The limits on allowed links are described in the *NonStop TS/MP System Management Manual*.
Links to server processes	Maximum of 1750 concurrent links per processor
Nowait depth	Maximum of 255 for nowait Pathsend procedure calls
PATHMON processes	Maximum of 256 PATHMON processes with which a LINKMON process can communicate
Pathsend requesters	Maximum of 256 concurrently active Pathsend requesters per processor
Server classes	Maximum of 1024 server classes to which all requesters in a processor can have outstanding links
Server-class send operations	Maximum of 255 concurrent outstanding send requests per requester
	Maximum of 512 concurrent outstanding send requests per processor
TMF transactions	Maximum of 100 outstanding TMF transactions per requester. (This limit is imposed by the TMF subsystem.)

B Examples

This appendix shows the source code for two example programs designed to help you understand how to write Pathsend programs and Pathway servers. The following examples are included:

- A context-free Pathsend requester coded in TAL
- A context-free Pathsend nested server example coded in COBOL85

Edit files containing the source code for these examples are provided on the site update tape (SUT). The source code for the Pathsend requester is in the file BREQS; the source code for the nested server is in the file PATHSRVS.

The examples allow you to see some of the programming concepts, described in previous sections of this manual, put into practice. These programs, however, require several other files provided on the SUT to run. You should read the file README (located on the SUT) carefully before attempting to run these programs.

Note. The program examples included in this manual and on the SUT are for reference purposes only and are intended to illustrate usage of Pathsend procedures. The program examples are not intended to form the basis of production programs. The examples do not demonstrate the definitive way to write Pathsend programs. There are other methods that might be more suitable for your requirements or application.

Pathsend Requester Example

Example B-1, BREQ, is a Pathsend context-free requester program coded in TAL. BREQ issues nowait SERVERCLASS_SEND_ calls to two server classes. The program demonstrates how to write a Pathsend program that issues nowait SERVERCLASS_SEND_ calls to two servers and waits for successful completion of both send operations before committing the transaction.

Example B-1. Context-Free Pathsend Requester Program

```
?PAGE "BROADCAST REQUESTER (BREQ) OVERVIEW"
  ! @@@ START COPYRIGHT @@@
  !  Tandem Confidential:  Need to Know only
  !  Copyright (c) 1980-1985, 1987-1995, Tandem Computers Incorporated
  !  Protected as an unpublished work.
  !  All Rights Reserved.
  !
  !  The computer program listings, specifications, and documentation
  !  herein are the property of Tandem Computers Incorporated and shall
  !  not be reproduced, copied, disclosed, or used in whole or in part
  !  for any reason without the prior express written permission of
  !  Tandem Computers Incorporated.
  ! @@@ END COPYRIGHT @@@
?SETTOG 1
?IFNOT 1

BREQ is a single-threaded program.  It gets its input from the input
file, which contains data used to make two Pathsend sends.  In
addition to doing the two sends, BREQ writes the input record to the
message log file.  The two sends and the write are treated as one
logical transaction.

The processing flow is like this: read one record from the input file,
initiate the I/O nowaited, complete the I/O, and write the reply from
the two servers to the output file before reading the next record from
the input file.

An input record is made up of the following data for each Pathsend
send: either a PATHMON ASSIGN name or a PATHMON system and process
name, followed by a server class name.

The PATHMON ASSIGN names get passed to BREQ at startup.  If a trace
file ASSIGN is present, then BREQ outputs a msg after it does any of
the following:

    . READs one record from the input file
    . Initiates a SERVERCLASS SEND
    . Initiates a WRITE to the msg log file
    . Executes BEGINTRANSACTION
    . Executes ENDTRANSACTION
    . Executes ABORTTRANSACTION
    . CANCELs outstanding I/O
    . After an I/O completes successfully
    . After an I/O times out
    . After an I/O fails
    . When a transaction is retried
    . When max-retries is exceeded

BREQ does a SERVERCLASS_SEND_ to the first server, a WRITE to the
message log file, and a SERVERCLASS_SEND_ to the second server.  All
three I/O's are NOWAIT and initiated in this sequence.

The reply from each server named in the request msg is simply its
process id.  BREQ puts together the reply for each server and writes
it to the output file.
```

```
The following files are used by BREQ and must exist before running the
program:

    INFILE      -- each record is one transaction
    OUTFILE     -- the outcome of the transaction

    ERROR LOG   -- Entry sequenced; record length 132; not audited.
                   The physical file name is read from the ASSIGN
                   'ERROR-LOG-FILE'.

    MESSAGE LOG -- Entry sequenced; record length 132; audited.
                   The physical file name is read from the ASSIGN
                   'MESSAGE-LOG-FILE'.

    TRACE (OPTIONAL) -- Entry sequenced; record length 80; not
                        audited; may be a terminal, may not be a
                        spooler location ($S.#SOME.LOC).  The physical
                        file name is read from the ASSIGN
                        'TRACE-FILE'.

The following ASSIGNS passed in:

    ERROR-LOG-FILE
       -- Used to record error msgs.

    MESSAGE-LOG-FILE
       -- A write to this file is part of a transaction.  It contains
          request msgs received by BREQ from DRIVER.

    TRACE-FILE (OPTIONAL)
       -- Used to log certain READs, SENDs, and TMF operations.

There can also be up to 50 ASSIGNs to give logical names to PATHMONs
that may be referenced in the input record.

The following PARAM is passed:

    MAX-RETRIES
       -- BREQ uses this number to calculate the number of times
          it will try to recover from PATHSEND SEND failures.

Example run-line:

      RUN BREQ/ NAME $BREQ, IN infile, OUT outfile/

The procedures on the following pages are listed in alphabetical
order.

?ENDIF 1

?PAGE "COMPILER DIRECTIVES, GLOBALS, AND EXTDECS"
?INSPECT, SYMBOLS, NOCODE
?LIST

?DATAPAGES 64
```

```
?PAGE "VERSION PROCEDURE DECLARATION"

?SOURCE VersProc( PS^EXAMPLE^MODULE )

?PAGE "GLOBAL DECLARATIONS FOR BREQ"

!   The following are GLOBAL defines

DEFINE def      = DEFINE#;
DEF    lit      = LITERAL#;

DEF str         = STRING#;
DEF dbl         = INT(32)#;

?PAGE "STRUCTS BREQ"

!   These are the STRUCTS used when compiling the BREQ program.
!   Comments appears to the right of the code.

?PAGE "STRUCTURE NEEDED TO STARTUP A PROCESS"

?SECTION STARTUP

!   The following is used when starting up

STRUCT .CI^STARTUP (*);
BEGIN
  INT MSGCODE;
  STRUCT DEFAULT;
  BEGIN
    INT VOLUME [0:3],
        SUBVOL [0:3];
    END;
  STRUCT INFILE;
  BEGIN
    INT VOLUME [0:3],
        SUBVOL [0:3],
        DNAME  [0:3];
    END;
  STRUCT OUTFILE;
  BEGIN
    INT VOLUME [0:3],
        SUBVOL [0:3],
        DNAME  [0:3];
    END;
  STRING PARAM [0:564];
  END; ! CI STARTUP
```

```
?PAGE "STRUCTURE NEEDED TO PROCESS ASSIGN MSGS"

?SECTION ASSIGN

!   The following is used to process ASSIGN msgs:

STRUCT CI^ASSIGN (*);                 !ASSIGN msg
BEGIN                                 !
  INT MSG^CODE;                       ![0] -2
  STRUCT LOGICALUNIT;                 !PARAMS to ASSIGN command
  BEGIN                               !
    STRING PROGNAMELEN,               ![1]
           PROGNAME[0:30],            !
           FILENAMELEN,               ![17]
           FILENAME[0:30];            !
  END;                                !
  INT(32) FIELDMASK;                  ![33] bit mask to indicate
                                      !which fields were supplied
                                      !(1=supplied):
                                      !  .<0> = TANDEM-FILENAME
                                      !  .<1> = PRI-EXT-SIZE
                                      !  .<2> = SEC-EXT-SIZE
                                      !  .<3> = FILE-CODE
                                      !  .<4> = EXCLUSION-SIZE
                                      !  .<5> = ACCESS-SPEC
                                      !  .<6> = RECORD-SIZE
                                      !  .<7> = BLOCK-SIZE
                                      !
  STRUCT TANDEMFILENAME;              ![35] TANDEM-FILENAME
  BEGIN                               !
    INT VOLUME[0:3],                  !
        SUBVOL[0:3],                  !
        DFILE [0:3];                  !
  END;                                !
                                      !
  !CREATESPEC                         !
                                      !
  INT PRIMARYEXTENT,                  ![47]
      SECONDARYEXTENT,                !
      FILECODE,                       !
      EXCLUSIONSPEC,                  ![50]   %00 if shared
                                      !       %20 if exclusive
                                      !       %60 if protected
                                      !
      ACCESSSPEC,                     ![51]   %0000 if I/O
                                      !       %2000 if input
                                      !       %4000 if output
                                      ![50-51] correspond to flag PARAM
                                      !of OPEN
                                      !
      RECORDSIZE,                     ![52]
      BLOCKSIZE;                      ![53]
END; ! ci ASSIGN !                    !msg size = 108 bytes
```

```
?PAGE "STRUCTURE NEEDED TO PROCESS PARAM MSGS"

?SECTION PARAM

!   The following is used to process PARAM messages

STRUCT CI^PARAM (*);                !PARAM msg
BEGIN                               !
  INT MSG^CODE,                     ![0] -3
      NUM^PARAMS;                   ![1] number of PARAMS in this msg
  STR PARAMETERS[0:1023];           ![2] PARAMS
END;                                !

?PAGE "STRUCTURE OF THE INPUT FILE"

?SECTION BREQ^INPUT

!   The following is the record format of the edit file, which is the
!   input to the BREQ program.
!

STRUCT BREQ^INPUT^REC^TEMPLATE (*);
BEGIN
  STRUCT SERVER^REQUEST [0:1];
  BEGIN
    STR PATHMON^ASSIGN^NAME[0:30];
    STR PATHMON^SYSTEM^AND^PROCESS^NAME[0:14];
    STR SERVER^CLASS[0:14];
  END;
END;

?PAGE "STRUCTURE OF THE BREQ OUTPUT RECORD"

?SECTION BREQ^OUTPUT

!   The following is the structure of the output record that BREQ
!   writes to the OUT file specified in the run line.

STRUCT BREQ^OUTPUT^REC^TEMPLATE (*);
BEGIN
  STRUCT SERVER^REPLY [0:1];
  BEGIN
    STR SYSTEM^NAME[0:7];
    STR PROCESS^NAME[0:7];
    STRUCT ERROR^MSG;
    BEGIN
      STR PATHSEND^ERROR[0:77];
      STR FILE^SYSTEM^ERROR[0:77];
    END;
  END;

  STR NON^SEND^ERROR^MSG[0:77];
END; ! BREQ output rec template
```

```
?PAGE "STRUCTURE OF A REQUEST TO PATHSRV"
?SECTION PATHSRV^REQUEST

!   The following is the format of the msg to PATHSRV.  It is
!   used in BREQ.

STRUCT PATHSRV^REQUEST^TEMPLATE (*);
BEGIN
  STR PATHMON^ASSIGN^NAME[0:30];
  STR PATHMON^SYSTEM^AND^PROCESS^NAME[0:14];
  STR SERVER^CLASS[0:14];
END;

?PAGE "STRUCTURE OF A REPLY FROM PATHSRV"
?SECTION PATHSRV^REPLY

!   The following is the format of the reply from PATHSRV.  It is
!   used in BREQ.

STRUCT PATHSRV^REPLY^TEMPLATE (*);
BEGIN
  INT REPLY^CODE;
  STRUCT THIS^SERVER;
  BEGIN
    STR SYSTEM^NAME[0:7];
    STR PROCESS^NAME[0:7];
  END;

  STRUCT SUBSIDIARY^SERVER;
  BEGIN
    STR SYSTEM^NAME[0:7];
    STR PROCESS^NAME[0:7];
  END;

  STR TMF^ABORT^REQUIRED;

  STRUCT ERROR^MSG;
  BEGIN
    STR PATHSEND^ERROR[0:77];
    STR FILE^SYSTEM^ERROR[0:77];
  END;

  STR NON^SEND^ERROR^MSG[0:77];
 END; ! PATHSRV reply template
```

```
?PAGE "STRUCTURE OF THE CONTROL BLOCK"

?SECTION CONTROL^BLOCK

!   This structure is used to keep data about a Pathsend send or
!   message log write.  Control blocks (cb's) are allocated from the
!   upper 32K using GETPOOL.

!   For each transaction, which is driven by data from one record in
!   the input file, 3 cb's are allocated (one each for the Pathsend
!   sends, and one for the message log write) and linked together in a
!   linked list.  At transaction completion, the cb memory is returned
!   using PUTPOOL.

STRUCT control^block^template (*);
BEGIN
  INT type;               ! Pathsend cb or msg log cb (see types below)
  INT fnum;               ! file # of msg log file if cb type is msg log
  INT scsend^opnum = fnum; ! sc send op # if cb type is Pathsend
  INT record^number;       ! corresponding input file rec number

  INT io^buf [0: $MAX ($LEN (pathsrv^request^template),     ! I/O buf
                       $LEN (pathsrv^reply^template)) - 1]; ! for sends
                                                           ! and writes.
  STRUCT pathsend^req^buf (pathsrv^request^template) = io^buf;
                                                            ! for sends
  STRUCT pathsend^reply^buf (pathsrv^reply^template) = io^buf;
                                                            ! for sends

  INT .input^data^buf (breq^input^rec^template ) = io^buf;
                                                    ! for writes, a ptr

  STR .pathmon^system^and^process^name;      ! pointer into input buf
  STR .serverclass^name;                     ! pointer into input buf

  INT pathsend^error;                    ! if error on send
  INT file^system^error;                 ! if error on send or write

  UNSIGNED (1) io^posted;                ! if send or write outstanding
  UNSIGNED (1) error^is^retryable;           ! if we can retry i/o
  UNSIGNED (14) not^used;                    ! filler

  STR pathsend^error^msg [0:77];             ! text for pathsend error
  STR file^system^error^msg [0:77];          ! text for filesys error

  INT .EXT next^cb (control^block^template); ! ptr to next cb in list,
END;                                         ! must be last field in struct

!   Types of control blocks.  A cb can be used for a PATHSEND send or
!   used for a message log write.

LIT cb^type^pathsend = 1;
LIT cb^type^msg^log  = 2;
```

```
?PAGE "GLOBAL DECLARATIONS FOR BREQ"

!   The following are GLOBAL literals:

LIT nil^addr            = -1D;

LIT true                = -1;
LIT false               =  0;

LIT e^eof               =  -1; ! end of file
LIT e^security^violation=  48; ! an access denied

LIT open^read^only      = %2000;
LIT open^write^only     = %4000;

LIT open^shared         = %0;
LIT open^nowait^disk    = %1;

LIT retry^path^failures =  1; ! sync depth param in the open

!   The following are TRACE file literals:

LIT trace^read^input           =  0;
LIT trace^pathsend             =  1;
LIT trace^write^to^msg^log     =  2;
LIT trace^begin^transaction    =  3;
LIT trace^end^transaction      =  4;
LIT trace^abort^transaction    =  5;
LIT trace^cancel^request       =  6;
LIT trace^write^blank^line     =  7;
LIT trace^completed^io         =  8;
LIT trace^io^timed^out         =  9;
LIT trace^io^failed            = 10;
LIT trace^retry^transaction    = 11;
LIT trace^max^retries^exceeded = 12;
```

```
?PAGE "T9153 PATHWAY PATHSEND SC Errors"
!
!  The following is a list of all the Server Class Errors. PATHSEND has
!  error numbers 900 - 950 reserved.
!

LITERAL
   FEScFirstError                = 900,  ! First avail. PATHSEND Error.
   FEScInvalidServerClassName    = 900,  ! Invalid server class name.
   FEScInvalidPathmonName        = 901,  ! Invalid Pathmon process name.
   FEScPathmonConnect            = 902,  ! Error with Pathmon
                                         ! connection (eg. Open, I/O,
                                         ! etc.).
   FEScPathmonMessage            = 903,  ! Unknown message received from
                                         ! PATHMON.
   FEScServerLinkConnect         = 904,  ! Error with Server Link
                                         ! connection
                                         ! (eg. Open, I/O, etc. ).
   FEScNoServerAvailable         = 905,  ! No Server Available.
   FEScNoSendEverCalled          = 906,  ! the user called SC_SEND_INFO
                                         ! before ever calling SC_SEND_.
   FEScInvalidSegmentID          = 907,  ! The caller uses an extended
                                      ! segment id that is out of range.
   FEScNoSegmentInUse            = 908,  ! The caller supplied a ref.
                                         ! parameter that is an extended
                                         ! address, but doesn't have an
                                         ! extended segment in use.
   FEScInvalidFlagsValue         = 909,  ! The caller set bits in flags
                                         ! parameter that are reserved
                                         ! and must be 0.
   FEScMissingParameter          = 910,  ! A required parameter was not
                                         ! supplied.
   FEScInvalidBufferLength       = 911,  ! One of the buffer length
                                         ! parameters is invalid.
   FEScParameterBoundsError      = 912,  ! A reference parameter is out
                                         ! of bounds.
   FEScServerClassFrozen         = 913,  ! The Server Class is Frozen.
   FEScUnknownServerClass        = 914,  ! PATHMON does not recognize
                                         ! Server Class name.
   FEScPathmonShutDown           = 915,  ! Send denied because Pathmon
                                         ! is shutting down.
   FEScServerCreationFailure     = 916,  ! Send denied by PATHMON
                                         ! because of Server creation
                                         ! failure.
   FEScServerClassTmfViolation   = 917,  ! The Tmf Transaction mode of
                                         ! the Send does not match that
                                         ! of the ServerClass (eg.
                                         ! Requester Send has a TransId
                                         ! and the ServerClass is
                                         ! configured with TMF OFF).
   FEScOperationAborted          = 918,  ! Send operation aborted. See
                                         ! accompanying Guardian error
                                         ! for more information.
   FEScInvalidTimeoutValue       = 919,  ! The caller supplied an
                                         ! invalid timeout value.
   FEScPFSUseError               = 920,  ! The caller's PFS segment
                                         ! could not be accessed.
   FEScTooManyPathmons           = 921,  ! The max. number of Pathmons
                                         ! allowed has been exceeded.
   FEScTooManyServerClasses      = 922,  ! The maximum number of server
                                         ! classes has been exceeded.
   FEScTooManyServerLinks        = 923,  ! The maximum number of server
                                         ! links has been exceeded.
   FEScTooManySendRequests       = 924,  ! The maximum number of send
                                         ! requests has been exceeded.
   FEScTooManyRequesters         = 925,  ! The maximum number of allowed
```

```
                                         ! requesters has been exceeded.
   FEScLinkMonConnect           = 947,  ! Error with LINKMON connection
                                         ! (eg. Open, I/O, etc. ).
   FEScLastError                = 950;  ! Last avail. PATHSEND error.
                                         ! This is for checking ranges
                                         ! not currently returned.

!   Miscellaneous file system errors

LIT FEok                         = 0;
LIT FEInvalidOp                  = 2;
LIT FEBoundsErr                  = 22;
LIT FEMissparam                  = 29;
LIT FESCError                    = 233;

!   Various file numbers used globally

INT  term^fnum := -1;
INT  in^fnum := -1;
INT  out^fnum := -1;
INT  error^log^fnum := -1;
INT  msg^log^fnum := -1;
INT  trace^fnum := -1;
```

```
INT .my^processid[0:3]; ! used to preface error msgs with my pid

!   Define a global array to hold error msgs that aren't PATHSEND send
!   errors.  For example, reference parameter errors, AWAITIOX timeout
!   errors, and validation errors.

STR .global^non^pathsend^error^msg [0:77] := [78 * [" "]];

LIT max^assigns = 56;
STRUCT .assign^table (ci^assign) [0:max^assigns - 1];

INT assign^count := 0;                ! count of assigns in table

INT max^retries := -1;                ! #times to retry Pathsend failures

INT .edit^control^block[0: (40 + (1024 / 2)) - 1];

!   We use GETPOOL and PUTPOOL to manage memory in the upper 32K.
!   Here are some vars used by these procedures.
!
LIT HeadSize = 38D; ! 19 words
LIT PoolAddr = %200000D;              ! upper 32K

INT .EXT PoolHead := PoolAddr;        ! room for the header
INT .EXT pool := PoolAddr + HeadSize; ! start of pool

LIT PoolSize = 32D*1024D;             ! 32K bytes

?LIST,PAGE

?PAGE "BREQ PROGRAM EXTDECS"

?PAGE "PROCESS THE ASSIGN MSGS"

PROC assign^proc (rucb, passthru, assign^msg, msg^len, match) VARIABLE;
INT .rucb,
    .passthru,
    .assign^msg,
     msg^len,
     match;
EXTERNAL;

?PAGE "RETURN AN ENTRY FROM THE ASSIGN TABLE"
INT PROC get^assign (logical^name, len);
STR .logical^name;
INT  len;
EXTERNAL;

?PAGE "READ THE STARTUP AND ASSIGN MSGS, AND OPEN THE LOG FILE"
PROC initialize;
EXTERNAL;
```

```
?PAGE "START ONE I/O IN THE TRANSACTION"

INT PROC initiate^IO (cb);
INT .EXT cb (control^block^template);
EXTERNAL;

?PAGE "START A WRITE OF THE INPUT REQUEST RECORD TO THE MSG LOG"
INT PROC initiate^write^to^message^log (cb);
INT .EXT cb (control^block^template);
EXTERNAL;

?PAGE "REPORT AN IO ERROR"
PROC IO^error (fnum);
INT fnum;
EXTERNAL;

?PAGE "SEARCH THE CONTROL BLOCKS FOR OUTSTANDING I/O"
INT PROC io^outstanding (cb);
INT .EXT cb (control^block^template);
EXTERNAL;

?PAGE "PRINT A MESSAGE TO THE TERMINAL AND ABEND"
PROC abend^with^my^abend^msg;
EXTERNAL;

?PAGE "PROCESS THE PARAM MSGS"

PROC param^proc (rucb, buf, param^msg, msg^len, match) VARIABLE;
INT .rucb,
    .buf,
    .param^msg (ci^param),
     msg^len,
     match;
EXTERNAL;

?PAGE "READ FROM AN EDIT FILE"
INT PROC read^ (fnum, buf, read^count, error);
INT  fnum;
STR .buf;
INT  read^count;
INT .error;
EXTERNAL;

?PAGE "LOOK AT WHY SERVERCLASS_SEND_INFO_ FAILED"
PROC ServerClass^Send^Info^error (error);
INT error;
EXTERNAL;

?PAGE "SET UP THE CONTROL BLOCKS FOR ONE TRANSACTION"

DBL PROC setup^control^blocks (input^buf, record^number);
INT .input^buf (breq^input^rec^template);
INT  record^number;
EXTERNAL;

?PAGE "START A TMF TRANSACTION"

PROC start^the^tmf^transaction;
EXTERNAL;
```

```
?PAGE "PROCESS THE STARTUP MSG"
PROC startup^proc (rucb, buf, msg, msg^len, match) VARIABLE;
INT .rucb,
    .buf,
    .msg (ci^startup),
     msg^len,
     match;
EXTERNAL;

?PAGE "STORE DATA IN THE CONTROL BLOCK"

PROC store^control^block^info (cb, data^buf, cb^type, record^number);
INT .EXT cb (control^block^template);
STR .data^buf (pathsrv^request^template);
INT  cb^type;
INT  record^number;
EXTERNAL;

?PAGE "BE SURE AN INCOMING REQUEST IS VALID"
INT PROC validate^breq^request (input^rec);
INT .input^rec (breq^input^rec^template);
EXTERNAL;

?PAGE "GET MY NAME OR PROCESS ID"
PROC who^am^i;
EXTERNAL;

?PAGE "WRITE TO THE TRACE FILE"
PROC write^trace^file (function, record^number) VARIABLE;
INT  function;
INT  record^number;
EXTERNAL;

?LIST

?NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS0 (ABEND, ABORTTRANSACTION,
?                                       AWAITIOX, BEGINTRANSACTION,
?                                       CANCELREQ, CLOSE, DEBUG,
?                                       DEVICEINFO, DNUMOUT, EDITREAD,
?                                       EDITREADINIT, ENDTRANSACTION,
?                                       FILEINFO, FNAMECOLLAPSE,
?                                       FNAMEEXPAND, GETCRTPID,
?                                       GETTRANSID, INITIALIZER,
?                                       MYPID, NEWPROCESS, NUMIN,
?                                       NUMOUT, OPEN, PROCESSORSTATUS,
?                                       READ, READUPDATE, REPLY,
?                                       SETMODE, STOP, WRITE,
?                                       WRITEREAD, WRITEX,
?                                       DEFINEPOOL, GETPOOL, PUTPOOL,
?                                       MYTERM,
?                                       SERVERCLASS_SEND_ ,
?                                       SERVERCLASS_SEND_INFO_ )

?LIST
```

```
!   The following dummy procedure is for example program version control

PS^EXAMPLE^VERSION^PROC;

?PAGE "PRINT A MESSAGE TO THE TERMINAL AND ABEND"


!   Many procedures check for error conditions that, under normal
!   circumstances, should not happen.  In these cases, before
!   ABENDing, a procedure will call this procedure to write a message
!   to the home terminal.  This proc calls ABEND.

PROC abend^with^my^abend^msg;

BEGIN
INT .filename [0:11];
INT  term^fnum;
INT .buf [0:79];
STR .sbuf := @buf '<<' 1;
STR .sp;

CALL MYTERM (filename);

CALL OPEN (filename, term^fnum);
IF <>
  THEN ! print an error msg and abend
    CALL IO^error (term^fnum);

sbuf ':=' "INTERNAL ERROR DETECTED, PROGRAM ABENDING" -> @sp;

CALL WRITE (term^fnum, buf, @sp '-' @sbuf);
IF <
  THEN ! print an error msg and abend
    CALL IO^error (error^log^fnum);

CALL CLOSE (term^fnum);

CALL ABEND;

END; ! PROC abend^with^my^abend^msg

?PAGE "ABORT THE TRANSACTION"


!
!   Abort the transaction I started
!
PROC abort^tmf^transaction;

BEGIN
INT tmf^error;
INT .buf [0:79];
STR .sbuf := @buf '<<' 1;
STR .sp;
```

```
IF (tmf^error := ABORTTRANSACTION)
  THEN ! failed to abort, file sys error in tmf^error
    BEGIN
    sbuf ':=' "ERROR WITH ABORTTRANSACTION: " -> @sp;
    CALL NUMOUT (sp, tmf^error, 10, 3);
    @sp := @sp[3];

    CALL WRITE (error^log^fnum, buf, @sp '-' @sbuf);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (error^log^fnum);

    CALL ABEND;
    END;

!   Successfully aborted the transaction

CALL write^trace^file (trace^abort^transaction);

sbuf ':=' "TRANSACTION ABORTED NORMALLY" -> @sp;

CALL WRITE (error^log^fnum, buf, @sp '-' @sbuf);
IF <
  THEN ! print an error msg and abend
    CALL IO^error (msg^log^fnum);

END; ! PROC abort^tmf^transaction

?PAGE "FIND OUT WHY SERVERCLASS_SEND_ FAILED"

!   The following procedure is called from PROCs complete^io and
!   initiate^IO. That is, SERVERCLASS_SEND_ and AWAITIOX can both
!   return file system error 233 (FESCERROR).

!   This proc gets detailed information about error 233 by calling
!   SERVERCLASS_SEND_INFO_ , which returns the pathsend error and
!   the file system error.

PROC analyze^send^error^233 (cb);
INT .EXT cb (control^block^template);

BEGIN
STR .EXT sp;
STR .EXT st;
INT  error;              ! used in call to serverclass_send_info_
INT  pathsend^error;     ! used in call to serverclass_send_info_
INT  filesystem^error;   ! used in call to serverclass_send_info_

!   Default to a non-retryable error

cb.error^is^retryable := false;
```

```
IF (error := SERVERCLASS_SEND_INFO_ (pathsend^error, filesystem^error))

  THEN ! problem making the call to SERVERCLASS_SEND_INFO_
    BEGIN
    CALL serverclass^send^info^error (error);
    RETURN;
    END;

!   Save off the errors and error msg text

cb.pathsend^error := pathsend^error;
cb.file^system^error := filesystem^error;

!   This application treats Pathsend errors 916, 918, and 924 as
!   retryable.  Other errors may be retryable, depending on your
!   application.  They generally reflect PATHWAY configuration
!   problems, and would be retryable after a delay.  For example,

!     904 (FEScServerLinkConnect): error with server link connection
!     (Open, I/O, etc.).

!     905 (FEScNoServerAvailable):  no server in the requested server
!     class has a free link.  the maximun number of links is specified
!     in PATHCOM by the server class attribute MAXLINKS.  A link could
!     become available if a TCP was shutdown.

!     913 (FEScServerClassFrozen): the server class is frozen.

!     916 (FEScServerCreationFailure): send denied by PATHMON because
!     of a server creation failure.

!     918 (FEScSendOperationAborted): the send operation was
!     terminated.  See file system error for more information.  For
!     example, corresponding file system error could be error 40,
!     time out.

!     924 (FEScTooManySendRequests): the maximum number of allowed
!     send requests has been exceeded.

IF (pathsend^error = FEScServerCreationFailure !916!
     OR pathsend^error = FEScOperationAborted !918!
      OR pathsend^error = FEScTooManySendRequests !924!)

  THEN ! we'll retry the SERVERCLASS_SEND_
    cb.error^is^retryable := true

  ELSE ! store error msg text, trap parameter errors
    BEGIN
    cb.pathsend^error^msg ':=' "PATHSEND ERROR: " -> @sp;
    CALL DNUMOUT (sp, $UDBL (pathsend^error), 10, 3);
    @sp := @sp[3];

    cb.file^system^error^msg ':=' "FILE SYSTEM ERROR: " -> @st;
    CALL DNUMOUT (st, $UDBL (filesystem^error), 10, 3);
    @st := @st[3];

    !   Use the pathsend error to check for parameter errors, and flag
    !   them as non-retryable.  Alternatively, we could use the value
    !   returned in the file system error to trap a bad param (error 2:
    !   FEInvalidOp, error 21: FEBadCount, error error 22: FEBadParam,
    !   error 29: FEMissParam).
    !
    !   The pathsend error does provide more specific information about
    !   about the problem.  For example, pathsend errors 907, 908, 909,
    !   and 919 all return the same file system error (error 2,
    !   FEInvalOp).
```

```
     IF pathsend^error = FEScInvalidSegmentId       !907!
       OR pathsend^error = FEScNoSegmentInUse       !908!
       OR pathsend^error = FEScInvalidFlagsValue    !909!
       OR pathsend^error = FEScMissingParameter     !910!
       OR pathsend^error = FEScInvalidBufferLength  !911!
       OR pathsend^error = FEScParameterBoundsError !912!
       OR pathsend^error = FEScInvalidTimeoutValue  !919!

       THEN ! the error returned from serverclass_send_ was a param error
         sp ':=' "(BAD PARAMETER PASSED TO SERVERCLASS_SEND_)" -> @sp;

     !   Include a msg if the file system error is a security violation

     IF filesystem^error = e^security^violation !48!
       THEN ! add error text for the user
         st ':=' "(SECURITY VIOLATION)";

     END; ! else trap parameter errors

END; ! PROC analyze^send^error^233

?PAGE "PROCESS THE ASSIGN MESSAGES"


!   This procedure is called by the GUARDIAN 90 PROC INITIALIZER from
!   PROC initialize.  It saves ASSIGN msgs, passed from our ancestor
!   TACL, in our ASSIGN table, and is called once for each ASSIGN msg
!   passed.

!   A count of the number of ASSIGNs in the table is kept in global
!   var assign^count, and is used in table lookups.

PROC assign^proc (rucb, passthru, assign^msg, msg^len, match) VARIABLE;
INT .rucb,
    .passthru,
    .assign^msg,
     msg^len,
     match;

BEGIN
INT .buf [0:79];
STR .sbuf := @buf '<<' 1;
STR .sp;
```

```
IF assign^count >= max^assigns
  THEN ! will exceed the size of our assign table
    BEGIN
    sbuf ':=' "NUMBER OF ASSIGNS EXCEEDS MAX^ASSIGNS" -> @sp;

    CALL WRITE (term^fnum, buf, @sp '-' @sbuf);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (term^fnum);

    CALL ABEND;
    END;

assign^table [assign^count] ':=' assign^msg FOR (msg^len + 1) / 2;
assign^count := assign^count + 1;

END; ! PROC assign^proc

?PAGE "CANCEL ALL OUR OUTSTANDING I/O"


!   This proc is called to cancel I/O posted but not completed.  We
!   use the io^posted flag in the control block to get the state of
!   the I/O.

!   This proc is called when one I/O in a transaction failed, either
!   at initiation time or completion time.

PROC cancel^outstanding^io (cb^list^head);
INT .EXT cb^list^head (control^block^template); ! ptr to head of cb list

BEGIN
INT .EXT cb (control^block^template) := @cb^list^head;

WHILE @cb <> nil^addr DO
  BEGIN

  IF cb.io^posted
    THEN ! this cb has I/O that hasn't completed
      BEGIN
      CALL CANCELREQ (cb.fnum, @cb);
      IF <
        THEN ! print an error msg and abend
          CALL IO^error (cb.fnum);

      cb.io^posted := false;
      CALL write^trace^file (trace^cancel^request, cb.record^number);
      END;

  @cb := @cb.next^cb;

  END; ! while

END; ! PROC cancel^outstanding^io
```

```
?PAGE "COMPLETE THE OUTSTANDING I/O"

!   This PROC waits for I/O to complete and returns the address of the
!   control block associated with the completion.  It returns nil^addr
!   if I/O completed with an error, or nil^addr if there was no
!   completion (in the case of a timeout, error 40).

DBL PROC complete^io;

BEGIN
INT .EXT cb (control^block^template);
DBL  tag := -1D;
DBL  timelimit := 300D;            ! wait 3 seconds (0.01 sec units)
LIT  anyfnum = -1;                 ! wait on any file
INT  fnum;                         ! used in call to awaitiox
STR .EXT sp;
INT  error;
INT .buf [0:79];
STR .sbuf := @buf '<<' 1;
STR .st;

!   When waiting for any I/O to complete (fnum = -1) with an AWAITIO
!   time limit <> 0, I/O is considered complete and no longer
!   outstanding (whether an error is returned or not) in every case
!   except when a timeout (error 40) occurs.
!
!   When a completion occurs, the file number of the completed call
!   is returned in the <fnum> param, and tag is returned in <tag>.

fnum := anyfnum;

CALL AWAITIOX (fnum,,, tag, timelimit);
IF =
  THEN ! one I/O successfully completed
    BEGIN
    @cb := tag;

    !   Do some basic checks to make sure the control block is what we
    !   expect.  We expect the file number returned by AWAITIOX to be
    !   the same as the fle number we saved in the control block when
    !   the I/O was initiated.  And we expect the control block to be
    !   a valid type.

    IF cb.fnum <> fnum OR
        (cb.type <> cb^type^pathsend AND cb.type <> cb^type^msg^log)

      THEN ! assertion failed
        CALL abend^with^my^abend^msg;

    cb.io^posted := false;

    CALL write^trace^file (trace^completed^io, cb.record^number);
    RETURN @cb;
    END;
```

```
!   An error or warning occurred

CALL FILEINFO (fnum, error);
IF <>
  THEN ! couldn't access the ACB
    CALL abend^with^my^abend^msg;

!   Check for a timeout error

IF error = 40
  THEN ! AWAITIO timed out and no completion has occurred
    BEGIN
    sbuf ':=' "AWAITIO TIMED OUT (ERROR 40)" -> @st;

    CALL WRITE (error^log^fnum, buf, @st '-' @sbuf);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (error^log^fnum);

    CALL write^trace^file (trace^io^timed^out);

    !   Save the error msg so we can include it in the output record

    global^non^pathsend^error^msg ':=' sbuf FOR @st '-' @sbuf;

    RETURN nil^addr;
    END;

!   I/O completed with some error, find out what it is

@cb := tag;

!   We expect the file number returned by AWAITIOX to be the same as
!   the fle number we saved in the control block when the I/O was
!   initiated.

IF cb.fnum <> fnum
  THEN ! assertion failed
    CALL abend^with^my^abend^msg;

cb.io^posted := false;

CALL write^trace^file (trace^io^failed, cb.record^number);
```

```
!   Check for errors on the disk operation

CASE cb.type OF
  BEGIN

  cb^type^msg^log ->

    BEGIN

    !   I/O to the message log file failed

    cb.file^system^error := error;
    cb.file^system^error^msg ':=' "ERROR: " -> @sp;
    CALL DNUMOUT (sp, $UDBL (error), 10,3);
    @sp := @sp[3];
    sp ':=' "RETURNED FROM AWAITIO ON I/O TO THE MESSAGE LOG FILE";

    RETURN nil^addr;
    END;

  cb^type^pathsend ->

    BEGIN

    !   A nowaited ServerClass_Send_ failed

    IF error = FeSCError !file system error 233!
      THEN ! call ServerClass_Send_Info_ for Pathsend & filesys errors
       CALL analyze^send^error^233 (cb)

      ELSE        ! no other errors should be returned by AWAITIOX for
        CALL abend^with^my^abend^msg;                  ! Pathsend sends

    END;

  OTHERWISE ->

    !   There are no other types of control blocks, so

    CALL abend^with^my^abend^msg;

  END; ! case

RETURN nil^addr;

END; ! PROC complete^io
```

```
?PAGE "CREATE THE CONTROL BLOCK MEMORY POOL"

!   This procedure uses the standard memory management routines to
!   declare a memory pool starting at the upper 32K, with a size of
!   32K bytes.

!   Control blocks (cb's) are managed as a linked list, and new cb's
!   are allocated in get^control^block.

PROC create^the^memory^pool;

BEGIN
INT status;
INT .buf [0:79];
STR .sbuf := @buf '<<' 1;
STR .sp;

IF (status := DEFINEPOOL (PoolHead, pool, PoolSize))
  THEN ! some memory allocation problem
    BEGIN
    sbuf ':=' "DEFINEPOOL FAILED TO CREATE THE MEMORY POOL" -> @sp;

    CALL WRITE (error^log^fnum, buf, @sp '-' @sbuf);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (error^log^fnum);

    CALL ABEND;
    END;

END; ! PROC create^the^memory^pool

?PAGE "END THE TMF TRANSACTION"

!   End the transaction I started, and log the outcome to the error
!   log file.  If successful, write a record to the trace file.

PROC end^the^tmf^transaction;

BEGIN
INT tmf^error;
INT .buf [0:79];
STR .sbuf := @buf '<<' 1;
STR .sp;

IF (tmf^error := ENDTRANSACTION)
  THEN ! file sys error in tmf^error
    BEGIN
    sbuf ':=' "ERROR WITH ENDTRANSACTION: " -> @sp;
    CALL NUMOUT (sp, tmf^error, 10, 3);
    @sp := @sp[3];


    CALL WRITE (error^log^fnum, buf, @sp '-' @sbuf);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (error^log^fnum);

    CALL ABEND;
    END;

!   Successfully ended the transaction

CALL write^trace^file (trace^end^transaction);
```

```
sbuf ':=' "TRANSACTION ENDED NORMALLY" -> @sp;
CALL WRITE (error^log^fnum, buf, @sp '-' @sbuf);
IF <
  THEN ! print an error msg and abend
    CALL IO^error (msg^log^fnum);

END; ! PROC end^the^tmf^transaction

?PAGE "RETURN AN ENTRY FROM THE ASSIGN TABLE"

!   This procedure is passed a logical ASSIGN name and searchs the
!   ASSIGN table for the ASSIGN name.  If found it returns the address
!   of the table entry, otherwise false.

INT PROC get^assign (logical^name, len);
STR .logical^name;
INT  len;

BEGIN
INT i := 0;
INT .assign^ (ci^assign);

WHILE i < assign^count DO
  BEGIN
  @assign^ := @assign^table[i];

  IF assign^.logicalunit.filename = logical^name FOR len
      AND assign^.logicalunit.filenamelen = len

    THEN ! found matching ASSIGN
      RETURN @assign^;

  i := i + 1;
  END;

RETURN false;
END; ! INT PROC get^assign
```

```
?PAGE "ALLOCATE A NEW CONTROL BLOCK FROM THE MEMORY POOL"

!   This proc gets memory from the pool for a new control block (cb).
!   The caller passes in the head of a list of control blocks, and it
!   links the new cb to the end of that list.

!   If the list head has a nil address, then the new cb becomes the
!   first element of the list.

!   This proc returns the address of the new control block.  Before
!   returning, the new cb is zero'd.

DBL PROC get^control^block (cb^list^head);
INT .EXT cb^list^head (control^block^template);

BEGIN
INT .EXT cb (control^block^template) := @cb^list^head;
INT .EXT new^cb (control^block^template);
INT .buf [0:79];
STR .sbuf := @buf '<<' 1;
STR .sp;

@new^cb := GETPOOL (PoolHead, $UDBL ($LEN (cb)));

IF @new^cb = -1D
  THEN ! couldn't get memory
    BEGIN
    sbuf ':=' "GETPOOL FAILED TO GET MEMORY IN GET^CONTROL^BLOCK" -> @sp;

    CALL WRITE (error^log^fnum, buf, @sp '-' @sbuf);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (error^log^fnum);

    CALL ABEND;
    END;

!   If the caller passed in a list head with nil address, then make
!   the new cb the first element in the list.

IF @cb = nil^addr
  THEN ! the user passed an empty list
    @cb := @new^cb

  ELSE ! link the new cb to the end of the list
    BEGIN

    WHILE @cb.next^cb <> nil^addr DO
      @cb := @cb.next^cb;

    @cb.next^cb := @new^cb;
    @cb := @cb.next^cb;

    END;
```

```
!   Clean the new control block, and set the link to nil address

cb ':=' 0 & cb FOR (($LEN (cb) + 1) / 2) - 1;

@cb.next^cb := nil^addr;

RETURN @cb;

END; ! DBL PROC get^control^block

?PAGE "READ THE STARTUP AND ASSIGN MSGS, AND OPEN VARIOUS LOG FILES"

!   This procedure calls the GUARDIAN 90 PROC INITIALIZER, which opens
!   $RECEIVE and handles the protocol with our ancestor, TACL.
!   INITIALIZER calls procs local to this program to read the startup
!   msg, save the ASSIGN msgs, and save the param msg.

!   This proc then opens the error log file, message log file, and the
!   trace file.

PROC initialize;

BEGIN
STRUCT .startup^msg (ci^startup);
INT .buf [0:79];
STR .sp := @buf '<<' 1;
STR .st;
INT .filename [0:11];
INT .error^log^file[0:11];
INT .msg^log^file[0:11];
INT .trace^file[0:11];
INT .assign^ (ci^assign);
STR .assign^name[0:17] := [18*[" "]];
STR .logical^name[0:30];
INT  setmode^param1;
INT  devtype, phys^reclen;

!   Open the home terminal to report initialization errors

CALL MYTERM (filename);
CALL OPEN (filename, term^fnum);
IF <>
  THEN ! print an error msg and abend
    CALL IO^error (term^fnum);

!   Get my process id and read the startup msg, ASSIGNs and PARAMs

CALL who^am^i;

CALL INITIALIZER (!rucb!, startup^msg, startup^proc, param^proc,
assign^proc);
```

```
!   Verify that the ASSIGNs for msg log file, error log file, and
!   trace file are present in our ASSIGN table.

!   The msg log file contains each record read from the input file

logical^name ':=' "MESSAGE-LOG-FILE               ";

IF NOT (@assign^ := get^assign (logical^name, 16))
  THEN ! user forgot assign
    BEGIN
    sp ':=' "MISSING ASSIGN 'MESSAGE-LOG-FILE' " ->@st;

    CALL WRITE (term^fnum, buf, @st '-' @sp);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (term^fnum);

    CALL ABEND;
    END;

!   The logical ASSIGN name is in assign.logicalunit.filename and the
!   physical file name is in internal format in assign.tandemfilename.
!   When a user adds an ASSIGN and omits the $VOL and/or SUBVOL, these
!   fields are also omitted from the ASSIGN STRUCT.

!   By collapsing the file name to it's external format
!   ($VOL.SUBVOL.FN) and then expanding it to it's internal format
!   ("\246ANCHORSUBVOL  FILE    "), we let the GUARDIAN 90 PROCs do
!   the work of adding default values.

CALL FNAMECOLLAPSE (assign^.tandemfilename.volume, msg^log^file);
CALL FNAMEEXPAND (msg^log^file, msg^log^file, startup^msg.default);

!   The error log file is used to log errors generated by BREQ

logical^name ':=' "ERROR-LOG-FILE                 ";

IF NOT (@assign^ := get^assign (logical^name, 14))
  THEN ! user forgot assign
    BEGIN
    sp ':=' "MISSING ASSIGN 'ERROR-LOG-FILE' " ->@st;

    CALL WRITE (term^fnum, buf, @st '-' @sp);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (term^fnum);

    CALL ABEND;
    END;

CALL FNAMECOLLAPSE (assign^.tandemfilename.volume, error^log^file);
CALL FNAMEEXPAND (error^log^file, error^log^file, startup^msg.default);
```

```
!   The trace file contains a log of specific events that occurred in
!   BREQ, e.g., a PATHSEND send or ENDTRANSACTION.

logical^name ':=' "TRACE-FILE                     ";

IF (@assign^ := get^assign (logical^name, 10))
  THEN ! optional assign, use if included
    BEGIN
    CALL FNAMECOLLAPSE (assign^.tandemfilename.volume, trace^file);
    CALL FNAMEEXPAND (trace^file, trace^file, startup^msg.default);

    CALL OPEN (trace^file, trace^fnum, open^write^only LOR
               open^shared, retry^path^failures);
    IF <>
      THEN ! open failed
        BEGIN
        sp ':=' "FAILED TO OPEN TRACE FILE" ->@st;

        CALL WRITE (term^fnum, buf, @st '-' @sp);
        IF <
          THEN ! print an error msg and abend
            CALL IO^error (term^fnum);

        CALL IO^error (trace^fnum);
        END;

!   By sharing the same terminal as the CI, which uses BREAK, we
!   gain BREAK access to the term with SETMODE 12 (set file
!   type access, param 2 = 1). File access is BREAK access.

    CALL DEVICEINFO (trace^file, devtype, phys^reclen);

    IF devtype.<4:9> = 6 ! Type TERM!
      THEN ! Do a setmode so we can write to it.
        CALL SETMODE (trace^fnum, 12, 0, 1);

    END
  ELSE ! trace assign not present, so don't trace
    trace^fnum := -1;

CALL OPEN (error^log^file, error^log^fnum, open^write^only LOR
           open^shared, retry^path^failures);
IF <>
  THEN ! open failed
    BEGIN
    sp ':=' "FAILED TO OPEN ERROR LOG FILE" ->@st;

    CALL WRITE (term^fnum, buf, @st '-' @sp);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (term^fnum);

    CALL IO^error (error^log^fnum);
    END;
```

```
CALL DEVICEINFO (error^log^file, devtype, phys^reclen);

IF devtype.<4:9> = 6 !type term!
  THEN ! Do a setmode so we can write to an un-paused TERM.
    CALL SETMODE (error^log^fnum, 12, 0, 1);

CALL OPEN (msg^log^file, msg^log^fnum, open^write^only LOR open^shared
           LOR open^nowait^disk, retry^path^failures);
IF <>
  THEN ! open failed
    BEGIN
    sp ':=' "FAILED TO OPEN MESSAGE LOG FILE" -> @st;

    CALL WRITE (term^fnum, buf, @st '-' @sp);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (term^fnum);

    CALL IO^error (msg^log^fnum);
    END;

CALL CLOSE (term^fnum);

END; ! PROC initialize

?PAGE "START ONE I/O IN THE TRANSACTION"

!   This procedure uses data in the control block (cb) to start a
!   Pathsend send or message log WRITE.  The I/O is nowaited, and the
!   I/O tag is the cb address.  If we fail to initiate the
!   SERVERCLASS_SEND_, then call analyze^send^error^233 to get the
!   associated Pathsend error and file system error.

!   This application treats initiation errors as non-retryable.

INT PROC initiate^IO (cb);
INT .EXT cb (control^block^template);

BEGIN
INT len;
INT ^ScSendOpNum;
INT pathmon^process^name^len;
INT serverclass^name^len;
INT flags := 0;
DBL timeout;
INT error;
STR .sp;

!   If this is a msg log control block then start the write to the msg
!   log file.

IF cb.type = cb^type^msg^log
  THEN ! start the disc op
    RETURN initiate^write^to^message^log (cb);
```

```
!   Start the Pathsend send to the server class.  Fill the send buf
!   with blanks so there's not a cascading Pathsend send from server
!   to server class.

cb.io^buf ':=' "  " & cb.io^buf FOR $OCCURS (cb.io^buf) - 1;

!   Set up the ServerClass_Send_ parameters:  get the PATHMON name and
!   server class name lengths, and set the values for <flags> and
!   <timeout>.

len := $OCCURS (cb.pathsend^req^buf.pathmon^system^and^process^name) - 1;

WHILE cb.pathmon^system^and^process^name [len] = " " OR
      cb.pathmon^system^and^process^name [len] = 0

  DO ! skip blanks and nulls
    len := len - 1;

pathmon^process^name^len := len + 1;

len := $OCCURS (cb.pathsend^req^buf.server^class) - 1;

WHILE cb.serverclass^name [len] = " " OR
      cb.serverclass^name [len] = 0

  DO ! skip blanks and nulls
    len := len - 1;

serverclass^name^len := len + 1;

!   Setting bit 15 in flags means send nowaited.  Setting timeout to
!   -1D means we're willing to wait forever for the send to complete,
!   and LINKMON will not cause this send to time out.

!   Remember, when a Pathsend send times out (AWAITIOX completes with
!   error 233 and ServerClass_Send_Info returns file system error 40),
!   the outstanding I/O to the server process is cancelled.

!   Note that setting a timeout value here is independent of setting a
!   timeout value in the call to AWAITIOX.

flags.<15> := 1;
timeout := -1D;

!   Initiate the Pathsend I/O.  If no error is returned from
!   SERVERCLASS_SEND_, then the I/O is successfully initiated.

!   If an error is returned, then the I/O was NOT successfully
!   initiated and NO I/O to a server process is outstanding.

!   NOTE: A value of -1 is returned in the <scsend-op-num> param for
!   waited sends.  A value of -1 is also returned for nowaited sends
!   that are not successfully initiated.
```

```
!   Also note that the <actual-reply-len> param is an optional
!   reference param that has a return value of 0 for nowaited I/O.

error := SERVERCLASS_SEND_ (cb.pathmon^system^and^process^name,

                            pathmon^process^name^len,
                            cb.serverclass^name,
                            serverclass^name^len,
                            cb.pathsend^req^buf,
                            $LEN (pathsrv^request^template),
                            $LEN (pathsrv^reply^template),
                            , ! actual reply len for waited I/O only !
                            timeout,
                            flags,
                            ^ScSendOpNum,
                            @cb !tag! );

IF error = FEok !0!
  THEN ! successfully initiated the Pathsend send
    BEGIN
    cb.ScSend^OpNum := ^ScSendOpNum;
    cb.io^posted := true;
    CALL write^trace^file (trace^pathsend, cb.record^number);
    RETURN true;
    END;

!   The send failed with an initiation error.  Find out
!   what happened.

cb.io^posted := false;

IF error = FESCError !file system error 233!
  THEN ! call SERVERCLASS_SEND_INFO to get Pathsend and file system errors
    CALL analyze^send^error^233 (cb)

  ELSE ! we should only see errors 0 or 233
    CALL abend^with^my^abend^msg;

RETURN false;

END; ! PROC initiate^IO
```

```
?PAGE "START A WRITE OF THE INPUT REQUEST RECORD TO THE MSG LOG"

!   This proc is called from initiate^IO, and starts the I/O to the
!   message log file.  The WRITE is tagged so we can identify this I/O
!   when completing it with AWAITIOX, or cancelling it with CANCELREQ.

!   If the WRITE fails, we will not return.

INT PROC initiate^write^to^message^log (cb);
INT .EXT cb (control^block^template);

BEGIN
INT error := 0;

CALL WRITEX (msg^log^fnum, cb.input^data^buf, $LEN(cb.input^data^buf),, @cb);
IF <
  THEN ! error occurred on the write
    BEGIN

    CALL FILEINFO (msg^log^fnum, error);
    IF <>
      THEN ! couldn't access the ACB
        CALL abend^with^my^abend^msg;

    !   Print an error message and abend

    CALL IO^error (msg^log^fnum);
    END;

!   I/O successfuly initiated

cb.io^posted := true;
cb.fnum := msg^log^fnum;

CALL write^trace^file (trace^write^to^msg^log, cb.record^number);

RETURN true;

END; ! INT PROC initiate^write^to^message^log

?PAGE "WRITE AN ERROR MSG TO THE ERROR LOG"

!   This procedure is passed a file number and gets the file system
!   error and file name associated with that file number.  It builds an
!   error msg and writes it to either the error log file or the home
!   terminal.  This PROC calls ABEND.

PROC IO^error (fnum);
INT fnum;

BEGIN
INT .buf [0:79];
STR .sbuf := @buf '<<' 1;
STR .sp;
INT .filename [0:11];
INT len;
INT error;
INT OutFnum;

!   If an OPEN failed, fnum is -1 and FILEINFO will return the error
!   associated with the failed OPEN.  The file name param, in this
!   case, will be invalid.

CALL FILEINFO (fnum, error, filename);
IF <>
  THEN ! couldn't access the ACB
```

```
    CALL abend^with^my^abend^msg;

sbuf ':=' "FILE SYSTEM ERROR " -> @sp;
CALL NUMOUT (sp, error, 10, 3);
@sp := @sp[3];

!   If the error is on a file already opened, include the file name
!   in the error msg.  If the error occurred while trying to open
!   a file, we can't include the file name.

IF fnum <> -1
  THEN ! the error occurred on a file already opened
    BEGIN
    sp ':=' " FILE " -> @sp;
    @sp := @sp [len := FNAMECOLLAPSE (filename, sp)];
    END;

!   Write the msg to the terminal if the error log isn't open

IF error^log^fnum <= 0
  THEN !  send output to the home term
    BEGIN
    CALL MYTERM (filename);
    CALL OPEN (filename, OutFnum);
    IF <> THEN CALL ABEND;
    END

  ELSE ! error log file is open
    OutFnum := error^log^fnum;

CALL WRITE (OutFnum, buf, @sp '-' @sbuf);
CALL CLOSE (OutFnum);

CALL ABEND;

END; ! PROC IO^error
```

```
?PAGE "SEARCH THE CONTROL BLOCKS FOR I/O POSTED BUT NOT COMPLETED"

!   I/O's were initiated out of control blocks.  Here we step through
!   the list of cb's and check for I/O that was initiated but not
!   completed.  If there are any outstanding I/O's, return true,
!   otherwise return false.

INT PROC io^outstanding (cb^list^head);
INT .EXT cb^list^head (control^block^template); ! first element in cb list

BEGIN
INT .EXT cb (control^block^template) := @cb^list^head;

WHILE @cb <> nil^addr
  DO
    BEGIN
    IF cb.io^posted
      THEN ! yes, there is outstanding I/O
        RETURN true;

    @cb := @cb.next^cb;
    END;

RETURN false;

END; ! PROC io^outstanding

?PAGE "OUTPUT THE RESULTS OF ONE TRANSACTION"

!   This procedure moves data from the two Pathsend control blocks
!   into the output buffer, and then writes the output buffer to the
!   out file.

PROC output^the^results (cb^list^head);
INT .EXT cb^list^head (control^block^template);

BEGIN
INT .EXT cb (control^block^template) := @cb^list^head;
STRUCT .out^buf (breq^output^rec^template);
INT i := 0;
STR .EXT sp;

!   Clear the output buffer

out^buf ':=' "  " & out^buf FOR ($LEN (out^buf) + 1) / 2;

!   Store the outcome of the two Pathsend sends into the output buffer

WHILE @cb <> nil^addr DO
  BEGIN

  IF cb.type = cb^type^pathsend
    THEN ! this cb has data about the outcome of a Pathsend send
      BEGIN
```

```
    !   Store the server's reply: system name and process name

    out^buf.server^reply[i].system^name ':='
      cb.pathsend^reply^buf.this^server.system^name
        FOR $OCCURS (cb.pathsend^reply^buf.this^server.system^name);

    out^buf.server^reply[i].process^name ':='
      cb.pathsend^reply^buf.this^server.process^name
        FOR $OCCURS (cb.pathsend^reply^buf.this^server.process^name);

    !   Store Pathsend error msg text and file system error msg text
    !   generated by this program.

    !   When an error is retryable we retry the transaction and
    !   don't include error msg text.  If the retries also failed,
    !   we need to add msg text now.

    IF (cb.pathsend^error AND NOT cb.pathsend^error^msg) OR
       (cb.file^system^error AND NOT cb.file^system^error^msg)

      THEN ! include error msg text
        BEGIN
        cb.pathsend^error^msg ':=' "PATHSEND ERROR: " -> @sp;
        CALL DNUMOUT (sp, $UDBL (cb.pathsend^error), 10, 3);

        cb.file^system^error^msg ':=' "FILE SYSTEM ERROR: " -> @sp;
        CALL DNUMOUT (sp, $UDBL (cb.file^system^error), 10, 3);

        END;

    IF cb.pathsend^error^msg
      THEN ! include msg text

        out^buf.server^reply[i].error^msg.pathsend^error ':='
          cb.pathsend^error^msg FOR $OCCURS (cb.pathsend^error^msg);

    IF cb.file^system^error^msg
      THEN ! include msg text

        out^buf.server^reply[i].error^msg.file^system^error ':='
     cb.file^system^error^msg FOR $OCCURS (cb.file^system^error^msg);

    !   Index to next server reply element in output buffer

    i := i + 1;

    END; ! if

@cb := @cb.next^cb;

END; ! while
```

```
!   Store non-Pathsend error messages generated by this program

out^buf.non^send^error^msg ':=' global^non^pathsend^error^msg
  FOR $OCCURS (out^buf.non^send^error^msg);

!   Write the record to the output file

CALL WRITE (out^fnum, out^buf, $LEN (out^buf));
IF <>
  THEN ! print an error msg and abend
    CALL IO^error (out^fnum);

END; ! PROC output^the^results

?PAGE "PROCESS THE PARAM MSGS"

!   This procedure is called by GUARDIAN 90 PROC INITIALIZER from PROC
!   initialize to save the PARAM MAX-RETRIES and store into variable
!   MAX-RETRIES.

PROC param^proc (rucb, passthru, param^msg, msg^len, match) VARIABLE;
INT .rucb;
INT .passthru;
INT .param^msg (ci^param);
INT  msg^len;
INT  match;

BEGIN
INT .buf [0:79];
STR .sbuf := @buf '<<' 1;
INT  i := 0, found := false;
STR .sp, .st;
INT status;
INT early^exit := false;

!   Unlike ASSIGNs, we receive all the PARAMs in a single
!   PARAM msg.

@sp := @param^msg.parameters;

!   The early^exit flag is used to exit the loop if the parameter has
!   an invalid value specified.  The found flag is used to exit the
!   loop if the param is found and has a valid value.

WHILE (i := i + 1) <= param^msg.num^params AND NOT found AND NOT early^exit
DO
  BEGIN

  IF sp[1] = "MAX-RETRIES"
    THEN ! found the param
      BEGIN
      sbuf ':=' sp[sp + 2] FOR sp[sp + 1] -> @st;
      st := 0;

      CALL NUMIN (sbuf, max^retries, 10, status);
      IF status
        THEN ! illegal value
          BEGIN
          max^retries := -1;
          early^exit := true;
          END

        ELSE
          found := true;

      END;
```

```
  IF NOT found AND NOT early^exit
    THEN ! look at the next param
      @sp := @sp[sp + sp[sp + 1] + 2];

  END; ! while i < num params and not found

IF NOT found
  THEN ! missing or bad param
    BEGIN
    sbuf ':=' "MISSING OR ILLEGAL PARAM 'MAX-RETRIES' " -> @sp;

    CALL WRITE (term^fnum, buf, @sp '-' @sbuf);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (error^log^fnum);

    CALL ABEND;

    END; ! if not found

END; ! PROC param^proc

?PAGE "PROCESS ONE TRANSACTION"

!   This PROC executes one transaction. It initiates the two nowaited
!   ServerClass_Sends_ and one message log write, and calls the
!   routine complete^IO to finish the I/O.

!   If the I/O completes successfully, this proc ends the transaction,
!   otherwise it aborts the transaction and cancels any outstanding
!   I/O.

INT PROC process^transaction (cb^list^head);
INT .EXT cb^list^head (control^block^template);

BEGIN
INT .EXT cb (control^block^template) := @cb^list^head;

CALL start^the^tmf^transaction;
```

```
!   Step through the list of control blocks (cb's), initiating the I/O
!   associated with each cb (a send or a write).

WHILE @cb <> nil^addr DO
  BEGIN

  IF NOT initiate^IO (cb)
    THEN ! failed to start the operation, treat error as non-retryable
      BEGIN
      CALL cancel^outstanding^io (cb^list^head);
      CALL abort^tmf^transaction;
      RETURN false;
      END;

  !   Successfully started the I/O for this cb, goto next cb

  @cb := @cb.next^cb;

  END; ! while

!   IO^Outstanding is a proc that steps through the list of cb's
!   checking for I/O that has been posted but not yet completed.

WHILE io^outstanding (cb^list^head) DO
  BEGIN

  !   Complete^IO returns @cb if the I/O completed successfully,
  !   otherwise it returns the nil address.

  @cb := complete^io;

  IF @cb = nil^addr
    THEN ! an I/O failed
      BEGIN
      CALL cancel^outstanding^io (cb^list^head);
      CALL abort^tmf^transaction;
      RETURN false;
      END;

  !   I/O completed successfully, check the reply buffer

  IF cb.type = cb^type^pathsend !pathsend control block!
      AND cb.pathsend^reply^buf.reply^code <> 0 !reply from server!

    THEN ! we expect the reply code to be 0
      CALL abend^with^my^abend^msg;

  END; ! while

!   Transaction completed successfully

CALL end^the^tmf^transaction;

RETURN true;

END; ! INT PROC process^transaction

?PAGE "READ FROM AN EDIT FILE"

!   This procedure reads one line from the IN file.
!   It returns the outcome of the read to the caller, with the
!   variable error containing the error associated with the read.

INT PROC read^ (fnum, buf, read^count, error);
INT  fnum;
STR .buf;
```

```
INT  read^count;
INT .error;

BEGIN
INT  status;
DBL  LineNum;

!   If status >= 0, the read was successful and status contains the
!   count of chars in the text line.  Actual bytes transferred <=
!   read^count.  If status < 0, an unrecoverable error occurred.

error := 0;

IF (status := EDITREAD (edit^control^block, buf, read^count, LineNum)) >= 0
  THEN ! read was successful
    RETURN true;

IF (error := status) = -1
  THEN ! end of file
    RETURN false;

!   Unexpected error reading the IN file, print an error msg and abend

CALL IO^error (fnum);

END; ! INT PROC read^

?PAGE "SEARCH THE CONTROL BLOCKS FOR RETRYABLE ERRORS"

!   Search the list of control blocks for errors that can be retried.
!   All Pathsend I/O that failed must be retryable (not just one send,
!   but both) for the transaction to be considered retryable.

!   When a transaction is retried, we re-use the existing list of
!   control blocks.

INT PROC retryable^transaction (cb^list^head);
INT .EXT cb^list^head (control^block^template); ! ptr to hd of cb list

BEGIN
INT .EXT cb (control^block^template) := @cb^list^head;
INT transaction^is^retryable := false;
```

```
WHILE @cb <> nil^addr
  DO
    BEGIN
    IF cb.type = cb^type^pathsend AND cb.pathsend^error
      THEN ! an error occurred on this Pathsend send

        BEGIN
        IF NOT cb.error^is^retryable
          THEN ! the transaction isn't retryable

            RETURN false;

        transaction^is^retryable := true;
        END;

    @cb := @cb.next^cb;
    END;

RETURN transaction^is^retryable;

END; ! INT PROC retryable^transaction

?PAGE "RETURN THE CONTROL BLOCK MEMORY TO THE POOL"

!   After we are done using all the control blocks in a list, return
!   the memory allocated for each control block.

PROC return^control^block^memory (cb^list^head);
INT .EXT cb^list^head (control^block^template);

BEGIN
INT .EXT cb (control^block^template) := @cb^list^head;
INT .EXT next^cb (control^block^template);
INT .buf [0:79];
STR .sbuf := @buf '<<' 1;
STR .sp;

WHILE @cb <> nil^addr DO
  BEGIN

  !   Save the address of the next element in the list before we
  !   return the current list element.

  @next^cb := @cb.next^cb;

  CALL PUTPOOL (PoolHead, cb);
  IF <
    THEN ! couldn't return the memory
      BEGIN
      sbuf ':=' "PUTPOOL FAILED TO RETURN MEMORY" -> @sp;

      CALL WRITE (error^log^fnum, buf, @sp '-' @sbuf);
      IF <
        THEN ! print an error msg and abend
          CALL IO^error (error^log^fnum);

      CALL ABEND;
      END; ! if

  @cb := @next^cb;
  END; ! while

END; ! PROC return^control^block^memory

?PAGE "LOOK AT WHY SERVERCLASS_SEND_INFO_ FAILED"
```

```
!   The following procedure is called from PROC analyze^send^error^233
!   when SERVERCLASS_SEND_INFO_ fails.  All these errors are
!   non-retryable, and all are programming errors.

PROC ServerClass^Send^Info^error (error);
INT error;

BEGIN

CASE error OF
  BEGIN

  FEInvalidOp !2!  ->

  !   Invalid segment in use or no segment in use and a param has xaddr

    global^non^pathsend^error^msg ':='
        "SERVERCLASS_SEND_INFO_ EXTENDED SEGMENT USAGE ERROR";

  FEBoundsErr !22! ->

  !   Param out of bounds

    global^non^pathsend^error^msg ':='
        "SERVERCLASS_SEND_INFO_ PARAMETER OUT OF BOUNDS";

  FEMissParam !29! ->

  !   A required param is missing

    global^non^pathsend^error^msg ':='
        "SERVERCLASS_SEND_INFO_ MISSING REQUIRED PARAMETER";

  OTHERWISE ->

  !   No other errors should be returned from ServerClass_Send_Info_

    global^non^pathsend^error^msg ':='
        "SERVERCLASS_SEND_INFO_ UNEXPECTED ERROR";

  END; ! case

END; ! PROC ServerClass^Send^Info^error
```

```
?PAGE "SET UP THE CONTROL BLOCKS FOR ONE TRANSACTION"

!   Here we get memory for each control block (cb) used in the
!   transaction, and we store data from the input record into the cb.
!   The data we store is data necessary to execute that part of the
!   transaction, a Pathsend send or msg log WRITE.

!   This proc returns the address of the first element in the list of
!   control blocks.

DBL PROC setup^control^blocks (input^buf, record^number);
INT .input^buf (breq^input^rec^template); ! input buffer
INT  record^number;                       ! input file rec num

BEGIN
INT .EXT cb^list^head (control^block^template) := nil^addr;
INT .EXT cb (control^block^template);

!   We pass into get^control^block the head of a list, and it creates
!   a new cb and links it onto the end of the list.  If we pass in a
!   list head with nil address, then list head becomes the first
!   element in the list.

@cb^list^head := get^control^block (cb^list^head);

!   Store data into this control block, used for the first Pathsend
!   send.

CALL store^control^block^info (cb^list^head, input^buf.server^request[0],
                               cb^type^pathsend, record^number);

!   Get the second element in the list, used for the WRITE to the msg
!   log file, and link it to the end of the list.

@cb := get^control^block (cb^list^head);

CALL store^control^block^info (cb, input^buf, cb^type^msg^log,
record^number);

!   Get the third element in the list, used for the second Pathsend
!   send, and link it to the end of the list.

@cb := get^control^block (cb^list^head);

CALL store^control^block^info (cb, input^buf.server^request[1],
                               cb^type^pathsend, record^number);

RETURN @cb^list^head;

END; ! DBL PROC setup^control^blocks
```

```
?PAGE "START THE TMF TRANSACTION"

!   The following procedure is called by the MAIN proc to start the
!   TMF transaction.

PROC start^the^tmf^transaction;

BEGIN
DBL trans^tag;
INT tmf^error;
INT .buf [0:79];
STR .sbuf := @buf '<<' 1;
STR .sp;

IF (tmf^error := BEGINTRANSACTION (trans^tag))
  THEN ! file system error returned in tmf^error
    BEGIN
    sbuf ':=' "ERROR WITH BEGINTRANSACTION: " -> @sp;
    CALL NUMOUT (sp, tmf^error, 10, 3);
    @sp := @sp[3];

    CALL WRITE (error^log^fnum, buf, @sp '-' @sbuf);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (error^log^fnum);

    CALL ABEND;
    END;

CALL write^trace^file (trace^begin^transaction);

END; ! PROC start^the^tmf^transaction

?PAGE "PROCESS THE STARTUP MSG"

!   This procedure is called from the GUARDIAN 90 PROC INITIALIZER
!   to save the startup msg, and OPEN the IN and OUT files.

PROC startup^proc (rucb, passthru, msg, msg^len, match) VARIABLE;
INT .rucb;
INT .passthru;
INT .msg (ci^startup);
INT  msg^len;
INT  match;

BEGIN
INT .buf [0:79];
STR .sbuf := @buf '<<' 1;
STR .sp;
INT FileCode;
INT status;

!   Pass the startup msg back because PROC initialize needs access
!   to the default volume and subvolume.

passthru ':=' msg for msg^len;

CALL OPEN (msg.infile, in^fnum, open^read^only);
IF <>
  THEN ! error opening the input file
    BEGIN
    sbuf ':=' "FAILED TO OPEN IN FILE" ->@sp;

    CALL WRITE (term^fnum, buf, @sp '-' @sbuf);
    IF <
      THEN ! print an error msg and abend
```

```
          CALL IO^error (term^fnum);

      CALL IO^error (in^fnum);
      END;

!  The IN file must be an EDIT file, so verify that now

CALL FILEINFO (in^fnum,,,,,,,,, FileCode);

IF FileCode <> 101
  THEN ! illegal file type
    BEGIN
    sbuf ':=' "INFILE MUST BE TYPE EDIT (101)" -> @sp;

    CALL WRITE (term^fnum, buf, @sp '-' @sbuf);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (term^fnum);

    CALL ABEND;
    END;

!   Setup the edit control block (a GLOBAL array) for calls
!   to EDITREAD.

IF (status := EDITREADINIT (edit^control^block, in^fnum, 1024))
  THEN ! print an error msg and abend
    CALL IO^error (in^fnum);

CALL OPEN (msg.outfile, out^fnum);
IF <>
  THEN ! error opening the output file
    BEGIN
    sbuf ':=' "FAILED TO OPEN OUT FILE" -> @sp;

    CALL WRITE (term^fnum, buf, @sp '-' @sbuf);
    IF <
      THEN ! print an error msg and abend
        CALL IO^error (term^fnum);

    CALL IO^error (out^fnum);
    END;

END; ! PROC startup^proc
```

```
?PAGE "STORE DATA IN THE CONTROL BLOCK"

!   This proc stores data to make one Pathsend send or one write to
!   the message log file.  The data is from one input record, and is
!   stored in the control block that's passed into this procedure.

PROC store^control^block^info (cb, data^buf, cb^type, record^number);
INT .EXT cb (control^block^template);
STR .data^buf (pathsrv^request^template);
INT  cb^type;
INT  record^number;

BEGIN
IF cb^type <> cb^type^pathsend AND cb^type <> cb^type^msg^log
  THEN ! assertion failed
    CALL abend^with^my^abend^msg;

cb.type := cb^type;
cb.record^number := record^number;

IF cb.type = cb^type^pathsend
  THEN ! store data necessary to do a Pathsend send
    BEGIN
    @cb.pathmon^system^and^process^name :=
        @data^buf.pathmon^system^and^process^name;

    @cb.serverclass^name := @data^buf.server^class;
    END

  ELSE ! the cb is for a write to msg log
    BEGIN
    @cb.input^data^buf := @data^buf '>>' 1;
    END;

END; ! PROC store^control^block^info

?PAGE "VALIDATE A REQUEST RECORD"

!   A valid request is in the format of breq^input^rec^template, and
!   contains data needed to make 2 Pathsend sends.  The data is a
!   PATHMON name (either explicit or an ASSIGN) and a server class
!   name.

!   If an ASSIGN is specified, then this proc looks up the ASSIGN name
!   in our ASSIGN table and puts the corresponding file name into the
!   PATHMON system and process name field of the input record.

!   This proc returns the outcome of the validation check to the
!   caller.

INT PROC validate^breq^input (input^rec);
INT .input^rec (breq^input^rec^template);

BEGIN
INT i;
INT .buf [0:79] := [80*["  "]];
STR .sbuf := @buf '<<' 1;
STR .sp;
STR .logical^name[0:30];
INT .assign^ (ci^assign);
```

```
?PAGE "SUBPROC TO WRITE A VALIDATION ERROR MSG"
  !
  !   If a record is invalid, write an error msg to the error log file
  !
  SUBPROC output^msg;

  BEGIN

  CALL WRITE (error^log^fnum, buf, @sp '-' @sbuf);
  IF <
    THEN ! print an error msg and abend
      CALL IO^error (error^log^fnum);

  !   Save the error msg so we can include it in the output record

  global^non^pathsend^error^msg ':=' sbuf FOR @sp '-' @sbuf;

  END; ! SUBPROC output^msg

?PAGE "BEGIN BODY OF VALIDATE BREQ INPUT"

FOR i := 0 to 1 DO
  BEGIN

  IF input^rec.server^request[i].server^class = [15*[" "]]
    THEN ! Invalid request
      BEGIN
      sbuf ':=' "INVALID REQUEST: SERVER CLASSES MUST BE SPECIFIED" ->
        @sp;
      CALL output^msg;
      RETURN false;
      END;

  IF (input^rec.server^request[i].pathmon^assign^name = [31*[" "]] AND
   input^rec.server^request[i].pathmon^system^and^process^name =
   [15*[" "]])
    THEN ! invalid request
      BEGIN
      sbuf ':=' "INVALID REQUEST: ASSIGN NAME " -> @sp;
      sp ':=' "OR SYSTEM AND PROCESS NAME MUST BE SPECIFIED" -> @sp;

      CALL output^msg;
      RETURN false;
      END;


  IF (input^rec.server^request[i].pathmon^assign^name <> [31*[" "]] AND
   input^rec.server^request[i].pathmon^system^and^process^name <>
   [15*[" "]])

    THEN ! invalid request
      BEGIN
      sbuf ':=' "INVALID REQUEST: BOTH ASSIGN AND SYSTEM " -> @sp;
      sp ':=' "AND PROCESS NAMES CAN'T BE SPECIFIED" -> @sp;

      CALL output^msg;
      RETURN false;
      END;

!   No format errors detected with server class names and PATHMON
!   ASSIGNs and process names.

  IF input^rec.server^request[i].pathmon^assign^name <> [31*[" "]]
   THEN ! try to get the ASSIGN name
     BEGIN
     logical^name ':=' input^rec.server^request[i].pathmon^assign^name
```

```
            FOR $OCCURS (input^rec.server^request.pathmon^assign^name);

      RSCAN logical^name[30] WHILE " " -> @sp;

      IF NOT (@assign^ := get^assign (logical^name, @sp[1] '-' @logical^name))
        THEN ! the ASSIGN isn't in our ASSIGN table
          BEGIN
          sbuf ':=' "ASSIGN '" -> @sp;
          sp ':=' logical^name FOR 31 -> @sp;
          RSCAN sp[-1] WHILE " " -> @sp;
          sp[1] ':=' "' NOT FOUND" -> @sp;

          CALL output^msg;
          RETURN false;
          END;

      !   Found the ASSIGN, save the PATHMON name

      CALL FNAMECOLLAPSE (assign^.tandemfilename.volume,
             input^rec.server^request[i].pathmon^system^and^process^name);

      END; ! look up the ASSIGN

   END; ! for loop

 RETURN true;

 END; ! INT PROC validate^breq^input
```

```
?PAGE "GET MY NAME OR PROCESS ID"
!
!   The following procedure gets my name or process id so that
!   error msg text can be preceded by my process name or cpu,pin.
!
PROC who^am^i;

BEGIN
STR .spid := @my^processid '<<' 1;

!   MYPID returns my cpu,pin and is input to GETCRTPID, which returns
!   my^processid in this form:
!     [0:2]       = process name or creation timestamp
!     [3].<4:7>  = cpu
!     [3].<8:15> = pin

CALL GETCRTPID (MYPID, my^processid);

IF spid <> "$"
  THEN ! I am not a named process, convert cpu,pin for output
    BEGIN
    CALL NUMOUT (spid, my^processid[3].<4:7>, 10, 2);
    spid[2] := ",";
    CALL NUMOUT (spid[3], my^processid[3].<8:15>, 10, 3);
    END;

END; ! PROC who^am^i

?PAGE "WRITE TO THE TRACE FILE"

!   This procedure prefaces each trace msg with this process's
!   process id, and does a waited WRITE to the trace file.  The trace
!   msg written depends on the function passed in.

PROC write^trace^file (function, record^number) VARIABLE;
INT  function;      ! what action we are tracing
INT  record^number; ! input file record number (optional param)

BEGIN
INT .write^buf[0:79] := [80*["  "]];
STR .swrite^buf := @write^buf '<<' 1;
STR .sp;
STR .spid := @my^processid '<<' 1;

IF NOT $PARAM (function)
  THEN ! missing required param
    CALL abend^with^my^abend^msg;

IF trace^fnum <= 0
  THEN ! user doesn't want to trace
    RETURN;
```

```
!   Let the user know who this msg is from

swrite^buf ':=' spid FOR 6 BYTES -> @sp;
RSCAN sp[-1] WHILE " " -> @sp;
@sp := @sp[1];
sp ':=' ": " -> @sp;

!   Include the record number

IF $PARAM (record^number)
  THEN ! include it in the msg
    BEGIN
    sp ':=' "RECORD #" -> @sp;
    CALL NUMOUT (sp, record^number, 10, 3);
    @sp := @sp[4];
    END;

CASE function OF
  BEGIN
   !0!  sp ':=' "READ ONE REQUEST MSG FROM INPUT FILE" -> @sp;
   !1!  sp ':=' "INITIATED ONE PATHSEND SEND" -> @sp;
   !2!  sp ':=' "INITIATED WRITE TO MESSAGE-LOG-FILE" -> @sp;
   !3!  sp ':=' "BEGIN TRANSACTION" -> @sp;
   !4!  sp ':=' "END TRANSACTION" -> @sp;
   !5!  sp ':=' "ABORT TRANSACTION" -> @sp;
   !6!  sp ':=' "CANCELREQ" -> @sp;
   !7!  @sp :=   @swrite^buf; ! write a blank line
   !8!  sp ':=' "I/O COMPLETED SUCCESSFULLY" -> @sp;
   !9!  sp ':=' "AWAITIO TIMED OUT" -> @sp;
  !10!  sp ':=' "I/O COMPLETED WITH AN ERROR" -> @sp;
  !11!  sp ':=' "RETRYING THE TRANSACTION" -> @sp;
  !12!  sp ':=' "MAX-RETRIES EXCEEDED, TRANSACTION ABORTED" -> @sp;

  !*!  OTHERWISE call abend^with^my^abend^msg;

  END; ! Case

CALL WRITE (trace^fnum, write^buf, @sp '-' @swrite^buf);
IF <
  THEN ! print an error msg and abend
    CALL IO^error (trace^fnum);

END; ! PROC write^trace^file
```

```
?PAGE "THE BREQ PROGRAM'S MAIN PROCEDURE"

!   This procedure calls the initialize routine to start up, sets up a
!   memory pool, and loops reading the IN file.

!   Input records are validated, control blocks are allocated, the
!   transaction is initiated and then completed, and control blocks
!   are de-allocated.

!   If we fail to execute the transaction and the error is retryable,
!   we retry it MAX-RETRIES times, where MAX-RETRIES is a param msg
!   defined at the TACL prompt.

PROC breq^program MAIN;

BEGIN
STRUCT .input^buf (breq^input^rec^template);     ! buf to hold input record
INT .EXT cb^list^head (control^block^template) := nil^addr;
                                                  !first cb in linked list.
INT error := 0;          ! used with read of input file
INT record^number := 0; ! input file record number we're processing
INT retry^count;         ! # times we retried transaction a failure

LIT global^msg^len = $OCCURS(breq^output^rec^template.non^send^error^msg);
                         ! len of non-Pathsend error msg buffer
CALL initialize;

!   Define the memory pool that we use to store the control blocks
!   (cb's), maintained as a linked list in the upper 32K.  Cb's are
!   allocated when we process one transaction, and are deallocated
!   when we complete one transaction.  A transaction is driven by data
!   in one record of the input file.

CALL create^the^memory^pool;

!   Loop until end of IN file, reading one record and doing the
!   transaction associated with that record.  A transaction is two
!   Pathsend sends and one WRITE to the message log file.

WHILE NOT error DO
  BEGIN
  input^buf ':=' "  " & input^buf FOR (($LEN (input^buf) + 1) / 2) - 1;

  global^non^pathsend^error^msg ':=' global^msg^len * [" "];

  IF read^ (in^fnum, input^buf, $LEN (input^buf), error)
    THEN ! successfully read 1 record
      BEGIN

      !   Record^Number is the input file record number we're
      !   processing, and is used in the output to associate the
      !   outcome of a transaction with the record used to generate
      !   the transaction.

      record^number := record^number + 1;

      !   Tracing is enabled by specifying a trace file ASSIGN.  If
      !   enabled, a trace record is written to the trace file when
      !   this program performs certain actions.  For example, reading
      !   an input record, and starting or aborting a transaction.

      CALL write^trace^file (trace^write^blank^line);
      CALL write^trace^file (trace^read^input, record^number);

      !   Besides validating the input record, the validate routine
      !   looks up ASSIGN names for PATHMON system and process names
```

```
!   specified in the input record, and puts the process name
!   associated with the ASSIGN into the PATHMON system and
!   process name field of the input record.

IF validate^breq^input (input^buf)
  THEN ! input record is valid
    BEGIN

    !   A valid request record is in the format of
    !   breq^input^rec^template, and contains data needed to
    !   make 2 Pathsend sends.  The data is a PATHMON name
    !   (either explicit or an ASSIGN) and a server class name.

    !   Allocate one control block (cb) each for the two
    !   Pathsend sends, and one cb for the message log write.
    !   The cb's are link listed together, and the head of the
    !   list is returned.  Data stored in the cb's is from the
    !   input record.

@cb^list^head := setup^control^blocks (input^buf, record^number);

    !   Process^transaction starts the tmf transaction,
    !   initiates the I/O, completes the I/O, and ends the tmf
    !   transaction.

    !   If an I/O fails, then process^transaction aborts the tmf
    !   transaction, cancels the outstanding I/O, and returns
    !   false.  In this case we will retry the transaction
    !   MAX-RETRIES times if the transaction is retryable.

    retry^count := 0;

    IF NOT process^transaction (cb^list^head)
      THEN ! the transaction failed
        BEGIN

        !   Retryable^transaction is a proc that returns true
        !   if all I/O that failed is retryable.


        WHILE retry^count < max^retries AND
            retryable^transaction (cb^list^head)
          DO
            BEGIN
            CALL write^trace^file (trace^retry^transaction);
            CALL process^transaction (cb^list^head);

            retry^count := retry^count + 1;
            END;

        If retry^count >= max^retries
          THEN
      ! the transaction failed after retrying max^retries times
            CALL write^trace^file (trace^max^retries^exceeded);

        END; ! if

    END; ! input record is valid

  !   Write the output to the OUT file

  CALL output^the^results (cb^list^head);

  !   Return all control block memory to the pool, and set the
  !   the first element in the list to nil address.
```

```
        CALL return^control^block^memory (cb^list^head);

        @cb^list^head := nil^addr;

        END; ! if read^

  END; ! while not error

IF error <> e^eof
  THEN ! print an error msg and abend
    CALL IO^error (in^fnum);

!   end of file input file

CALL CLOSE (in^fnum);
CALL CLOSE (out^fnum);
CALL CLOSE (msg^log^fnum);
CALL CLOSE (error^log^fnum);

IF trace^fnum > 0 THEN
  CALL CLOSE (trace^fnum);

CALL STOP;

END; ! PROC breq^program

?NOMAP
```

Nested Server Example

Example B-2, PATHSRV, is a context-free nested server coded in COBOL85. PATHSRV uses Pathsend calls within a server to access another server.

Example B-2. Context-Free Server Program

```
*    @@@ START COPYRIGHT @@@
*     Tandem Confidential:  Need to Know only
*     Copyright (c) 1980-1985, 1987-1995, Tandem Computers Incorporated
*     Protected as an unpublished work.
*     All Rights Reserved.
*
*     The computer program listings, specifications, and documentation
*     herein are the property of Tandem Computers Incorporated and shall
*     not be reproduced, copied, disclosed, or used in whole or in part
*     for any reason without the prior express written permission of
*     Tandem Computers Incorporated.
*    @@@ END COPYRIGHT @@@
?SYMBOLS

?SEARCH  $SYSTEM.SYSTEM.COBOLLIB
?CONSULT $SYSTEM.SYSTEM.COBOLEX0

?SAVE ASSIGNS 50, PARAM

 IDENTIFICATION DIVISION.
 PROGRAM-ID. PATHSRV.


*  THE COBOL85 SERVER PATHSRV READS A REQUEST FROM $RECEIVE AND
*  RETURNS ITS GUARDIAN PROCESS-ID AND THAT OF A SUBSIDIARY SERVER
*  CLASS (IF ANY) IDENTIFIED ON THE REQUEST MESSAGE. THE PATHMON OF
*  THE SUBSIDIARY SERVER CLASS IS IDENTIFIED IN THE REQUEST MESSAGE IN
*  ONE OF THREE WAYS AS FOLLOWS:
*
*     - BY AN ASSIGN NAME WHICH REFERENCES AN ASSIGN WITH THE
*       PATHMON SYSTEM AND PROCESS NAME.
*
*     - WITH AN EXPLICIT SYSTEM AND PROCESS NAME OF PATHMON.
*
*     - IF BOTH THE ASSIGN NAME AND THE EXPLICIT SYSTEM AND PROCESS
*       NAME IN THE REQUEST MESSAGE ARE BLANK THEN THE PATHMON FOR THE
*       SUBSIDIARY SERVER CLASS DEFAULTS TO BE THE SAME PATHMON AS
*       THAT WHICH OWNS THE SERVER MAKING THE PATHSEND SEND. THIS IS
*       POSSIBLE AS LONG AS THE SENDING SERVER IS NOT ASSOCIATIVE (IN
*       WHICH CASE ITS ANCESTOR MAY NOT BE A PATHMON).
*
*  THE FIRST AND THIRD OPTIONS DEMONSTRATE WAYS TO AVOID HARD
*  CODING A PATHMON SYSTEM AND PROCESS IN A PROGRAM THAT DOES A
*  PATHSEND SEND.
*
*  IN THE EVENT OF A PROBLEM AN ERROR MESSAGE AND A FLAG TO INDICATE
*  THAT A TMF ABORT IS REQUIRED ARE ALSO RETURNED TO THE REQUESTER.
*
```

```
*  ANY ERRORS THAT CAN NOT BE REPORTED BY PATHSRV BACK TO ITS
*  REQUESTER ARE WRITTEN TO AN ERROR LOG FILE. THIS ENTRY SEQUENCED
*  FILE WITH 132 BYTES RECORDS MUST EXIST PRIOR TO THE EXECUTION OF
*  PATHSRV. ITS ASSIGN NAME IS ERROR-LOG-FILE.


 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. TANDEM.
 OBJECT-COMPUTER. TANDEM.

 INPUT-OUTPUT SECTION.

 FILE-CONTROL.

    SELECT MESSAGE-IN-FILE
           ASSIGN TO $RECEIVE
           FILE STATUS IS WS-FILE-STATUS.

    SELECT MESSAGE-OUT-FILE
           ASSIGN TO $RECEIVE
           FILE STATUS IS WS-FILE-STATUS.

    SELECT ERROR-LOG-FILE
           ASSIGN TO "ERRORLOG"
           ORGANIZATION IS SEQUENTIAL
           ACCESS IS SEQUENTIAL
           FILE STATUS IS WS-FILE-STATUS.

 RECEIVE-CONTROL.

    TABLE OCCURS 10 TIMES
    SYNCDEPTH 1.

/
 DATA DIVISION.

 FILE SECTION.

 FD  MESSAGE-IN-FILE
     DATA RECORD IS PATHSRV-REQUEST.

 01  PATHSRV-REQUEST.
     03  SUBSIDIARY-SERVER.
         05  PATHMON-ASSIGN-NAME                   PIC X(31).
         05  PATHMON-SYSTEM-AND-PROCESS            PIC X(15).
         05  SERVER-CLASS                          PIC X(15).

 FD  MESSAGE-OUT-FILE
     DATA RECORD IS PATHSRV-REPLY.

 01  PATHSRV-REPLY.
     03  REPLY-CODE                                PIC S9(4) COMP.
     03  THIS-SERVER.
         05  SYSTEM-NAME                           PIC X(8).
         05  PROCESS-NAME                          PIC X(8).
     03  SUBSIDIARY-SERVER.
         05  SYSTEM-NAME                           PIC X(8).
         05  PROCESS-NAME                          PIC X(8).
     03  TMF-ABORT-REQUIRED                        PIC X.
     03  ERROR-MESSAGE.
         05  PATHSEND-ERROR                        PIC X(78).
         05  FILE-SYSTEM-ERROR                     PIC X(78).
     03  NON-SEND-ERROR-MESSAGE                    PIC X(78).

 FD  ERROR-LOG-FILE
```

```
    DATA RECORD IS ERROR-LOG-REC.

01  ERROR-LOG-REC                                  PIC X(132).


WORKING-STORAGE SECTION.

01  WS-FILE-STATUS                                 PIC X(2)  VALUE ZERO.
    88  WS-CLOSE-FROM-REQUESTER                    VALUE "10".

01  WS-FILE-SYSTEM-ERROR-MESSAGE.
    03  FILLER                                     PIC X(25)
        VALUE "FILE SYSTEM ERROR. FILE: ".
    03  WS-LOGICAL-FILE-NAME                       PIC X(31).
    03  FILLER                                     PIC X(13)
        VALUE "FILE STATUS: ".
    03  WS-FILE-STATUS-ERROR-MESSAGE               PIC 99.

01  WS-NUMERIC-DISPLAY                             PIC S9(9).


*  YOU CAN USE THE FOLLOWING WS-ASSIGN TABLE TO STORE UP TO FIFTY

*  ASSIGNS.  THESE ASSIGNS ARE FOR THE PATHMON SYSTEM AND PROCESS
*  NAMES OF SUBSIDIARY SERVERS. IF YOU USE PATHMON-ASSIGN-NAME IN THE
*  REQUEST MESSAGE (PATHSRV-REQUEST), THEN THE ASSOCIATED PATHMON
*  SHOULD BE IN THIS TABLE. THIS IS ONE WAY OF AVOIDING HARD-CODING
*  PATHMON SYSTEM AND PROCESS NAMES.

01  WS-ASSIGN-TABLE.
    03  WS-ASSIGN-NAME                       PIC X(31)  OCCURS 50.
    03  WS-SYSTEM-AND-PROCESS                PIC X(15)  OCCURS 50.
    03  WS-NUMBER-OF-ENTRIES                 PIC S9(4)  COMP  VALUE ZERO.
    03  WS-INDEX                             PIC S9(4)  COMP  VALUE ZERO.

*  THE FOLLOWING PARAMETERS ARE USED WHEN CALLING GETASSIGNTEXT

01  WS-GETASSIGNTEXT-PARAM.
    03  WS-PORTION                           PIC X(30).
    03  WS-TEXT                              PIC X(32).
    03  WS-MESSAGE-NUMBER                    PIC S9(4) COMP VALUE 1.
    03  WS-RESULT                            PIC S9(4) COMP VALUE ZERO.
```

```
*   PROCESSINFO IS CALLED TO SEE IF THE MOM OF THE SERVER PROCESS IS A
*   PATHMON. (THIS IDENTIFIES WHETHER THE SERVER IS ASSOCIATIVE.) AN
*   ERROR NUMBER (WS-ERROR) MY BE RETURNED FOR A CALL TO PROCESSINFO.

 01  WS-PROCESSINFO-PARAM.
     03  WS-ERROR                               PIC S9(4) COMP.


*   THE FOLLOWING PARAMETERS ARE USED IN A CALL TO LOOKUPPROCESSNAME,
*   WHICH LOCATES THE MOM OF THE CURRENT SERVER PROCESS (TO SEE IF IT'S
*   A PATHMON).

 01  WS-LOOKUPPROCESSNAME-PARAM.
     03  WS-PROCESS-NAME                        PIC X(6).
     03  FILLER                                 PIC X(4).
     03  WS-ANCESTOR-PROCESS-ID                 PIC X(8).


*   THE FOLLOWING PARAMETERS ARE USED IN A CALL TO SERVERCLASS_SEND_.


 01  WS-SERVERCLASS-SEND-PARAM.
     03  WS-ERROR                         PIC S9(4) COMP VALUE ZERO.
     03  WS-PATHMON-PROCESS-NAME          PIC X(15).
     03  WS-PATHMON-PROCESS-NAME-LEN      PIC S9(4) COMP VALUE 15.
     03  WS-SERVER-CLASS-NAME             PIC X(15).
     03  WS-SERVER-CLASS-NAME-LEN         PIC S9(4) COMP VALUE 15.
     03  WS-MESSAGE-BUFFER                PIC X(300).
     03  WS-REQUEST-LEN                   PIC S9(4) COMP VALUE 63.
     03  WS-MAXIMUM-REPLY-LEN             PIC S9(4) COMP VALUE 300.
     03  WS-ACTUAL-REPLY-LEN              PIC S9(4) COMP.
     03  WS-TIMEOUT                       PIC S9(9) COMP VALUE 12000.


*   THE FOLLOWING PARAMETERS ARE USED IN A CALL TO SERVERCLASS_SEND_INFO_


 01  WS-SERVERCLASS-SEND-INFO-PARAM.
     03  WS-ERROR                         PIC S9(4) COMP VALUE ZERO.
     03  WS-PATHSEND-ERROR                PIC S9(4) COMP.
     03  WS-FILE-SYSTEM-ERROR             PIC S9(4) COMP.


*   THE FOLLOWING PARAMETERS ARE USED IN A CALL TO MYPID AND MYCRTPID,
*   WHICH ARE USED TO IDENTIFY THE CURRENT PROCESS. THE PROCESS NAME IS
*   RETURNED TO THE REQUESTER IN THE PATHSRV-REPLY MESSAGE.

 01  WS-MY-PROCESS.
     03  WS-CPU-PIN                             PIC S9(4) COMP.
     03  WS-PROCESS-ID.
         05  WS-PROCESS-NAME                    PIC X(6).
         05  FILLER                             PIC X(2).
     03  WS-SYSTEM-NUMBER                       PIC 9(4) COMP.
     03  WS-SYSTEM-NAME                         PIC X(8).
```

```
*   THE FOLLOWING PARAMETERS ARE USED IN CALLS TO PROCESSINFO AND
*   MYSYSTEMNUMBER, WHICH ARE USED TO IDENTIFY THE MOM OF THE SERVER
*   PROCESS.

 01  WS-MOM-PROCESS.
     03  WS-PROCESS-ID-GENERIC.
         05  WS-BYTE-1                           PIC X.
             88  WS-LOCAL                        VALUE "$".
             88  WS-NETWORK                      VALUE "\".
         05  FILLER                              PIC X(5).
         05  WS-CPU-PIN                          PIC S9(4) COMP.
     03  WS-PROCESS-ID-LOCAL  REDEFINES WS-PROCESS-ID-GENERIC.
         05  WS-PROCESS-ID                       PIC X(6).
         05  WS-CPU-PIN                          PIC S9(4) COMP.
     03  WS-PROCESS-ID-NETWORK  REDEFINES WS-PROCESS-ID-GENERIC.
         05  WS-BACKSLASH                        PIC X.
         05  WS-SYSTEM-NUMBER-1-BYTE             PIC X.
         05  WS-PROCESS-ID                       PIC X(4).
         05  WS-CPU-PIN                          PIC S9(4) COMP.
     03  WS-SYSTEM-NUMBER                        PIC 9(4) COMP.
     03  WS-SYSTEM-NUMBER-2-BYTES REDEFINES WS-SYSTEM-NUMBER.
         05 FILLER                               PIC X.
         05 WS-BYTE-2                            PIC X.
     03  WS-PROGRAM-FILENAME.
         05  FILLER                              PIC X(16).
         05  WS-FILE                             PIC X(8).
     03  WS-SYSTEM-NAME                          PIC X(8).
     03  WS-PROCESS-NAME                         PIC X(6).
     03  WS-SYSTEM-AND-PROCESS                   PIC X(15).


*   THIS FLAG INDICATES WHETHER THE MOM OF THE SERVER PROCESS IS A
*   PATHMON. THE VALUE "N" MEANS THAT MOM IS A PATHMON, THE VALUE "Y"
*   MEANS THAT MON IS NOT A PATHMON, IN WHICH CASE THE PATHSRV-REQUEST
*   MESSAGE MUST IDENTIFY THE PATHMON WITH AN EXPLICIT NAME OR AN
*   ASSIGN. (THE DEFAULT TO THE SERVER PROCESS MOM CANNOT BE TAKEN.)

 01  WS-SERVER-IS-ASSOCIATIVE                    PIC X.


*   THE PATHSRV-REQUEST MESSAGE FROM THE REQUESTER IS VALIDATED IN THIS
*   PROGRAM. IF IT IS NOT VALID, THIS FLAG IS SET TO "N"

 01  WS-VALID-PATHSRV-REQUEST                    PIC X.
```

```
*   IF PATHSRV-REQUEST SPECIFIES A SERVER CLASS, THEN A PATHSEND
*   REQUEST IS MADE TO A SUBSIDIARY SERVER. THE FOLLOWING GROUP DATA
*   ITEMS ARE THE REQUEST AND REPLY MESSAGE LAYOUTS. THE REQUEST IS
*   ALWAYS SPACE FILLED BECAUSE THE SUBSIDIARY SERVER IS NEVER ASKED TO
*   MAKE A REQUEST OF ANOTHER SERVER.

 01  SUBSIDIARY-REQUEST.
     03  SUBSIDIARY-SERVER.
         05  PATHMON-ASSIGN-NAME                 PIC X(31).
         05  PATHMON-SYSTEM-AND-PROCESS          PIC X(15).
         05  SERVER-CLASS                        PIC X(15).

 01  SUBSIDIARY-REPLY.
     03  REPLY-CODE                              PIC S9(4) COMP.
     03  THIS-SERVER.
         05  SYSTEM-NAME                         PIC X(8).
         05  PROCESS-NAME                        PIC X(8).
     03  SUBSIDIARY-SERVER.
         05  SYSTEM-NAME                         PIC X(8).
         05  PROCESS-NAME                        PIC X(8).
     03  TMF-ABORT-REQUIRED                      PIC X.
     03  ERROR-MESSAGE.
         05  PATHSEND-ERROR                      PIC X(78).
         05  FILE-SYSTEM-ERROR                   PIC X(78).
     03  NON-SEND-ERROR-MESSAGE                  PIC X(78).


*   THE FOLLOWING FLAG INDICATES THE SUCCESS OR FAILURE OF A CALL TO
*   SERVERCLASS_SEND_. "Y" MEANS SUCCESS. "N" MEANS FAILURE.

 01  WS-SERVERCLASS-SEND-OKAY                    PIC X.


/
 PROCEDURE DIVISION.

 DECLARATIVES.


 DECL-MESSAGE-IN-FILE SECTION.
    USE AFTER ERROR PROCEDURE ON MESSAGE-IN-FILE.
    MOVE "MESSAGE-IN-FILE" TO WS-LOGICAL-FILE-NAME.
    MOVE WS-FILE-STATUS TO WS-FILE-STATUS-ERROR-MESSAGE.

 DECL-MESSAGE-OUT-FILE SECTION.
    USE AFTER ERROR PROCEDURE ON MESSAGE-OUT-FILE.
    MOVE "MESSAGE-OUT-FILE" TO WS-LOGICAL-FILE-NAME.
    MOVE WS-FILE-STATUS TO WS-FILE-STATUS-ERROR-MESSAGE.


 DECL-ERROR-LOG-FILE SECTION.
    USE AFTER ERROR PROCEDURE ON ERROR-LOG-FILE.
    MOVE "ERROR-LOG-FILE" TO WS-LOGICAL-FILE-NAME.
    MOVE WS-FILE-STATUS TO WS-FILE-STATUS-ERROR-MESSAGE.

 END DECLARATIVES.


 0000-MAINLINE.


*   THIS PARAGRAPH CONTAINS THE OVERALL LOGIC OF THE PROGRAM.  AFTER
*   INITIALIZATION, $RECEIVE IS READ. IF THE READ IS SUCCESSFUL, THE
*   REQUEST MESSAGE IS PROCESSED IN 0200-PROCESS-TRANSACTION AND THE
*   REPLY (PATHSRV-REPLY) IS WRITTEN TO $RECEIVE. $RECEIVE IS THEN READ
*   AGAIN. THE LOOP CONTINUES UNTIL A CLOSE IS RECEIVED FROM THE
```

```
*   REQUESTER ON $RECEIVE.
*
*   IF AN ERROR IS FOUND WHILE READING $RECEIVE, A MESSAGE IS WRITTEN
*   TO THE ERROR-LOG-FILE AND THE PROGRAM STOPS.

     PERFORM  0100-INITIALIZE.
     PERFORM UNTIL WS-CLOSE-FROM-REQUESTER
        READ MESSAGE-IN-FILE
        IF WS-FILE-STATUS = ZERO
           PERFORM 0200-PROCESS-TRANSACTION
           WRITE PATHSRV-REPLY
        ELSE
           IF NOT WS-CLOSE-FROM-REQUESTER
              MOVE WS-FILE-SYSTEM-ERROR-MESSAGE TO ERROR-LOG-REC
              PERFORM 9000-WRITE-ERROR-LOG-REC
              PERFORM 9900-STOP-RUN
           END-IF
        END-IF
     END-PERFORM
     PERFORM 0300-CLOSEDOWN
     PERFORM 9900-STOP-RUN.


  0100-INITIALIZE.
     OPEN EXTEND ERROR-LOG-FILE SHARED.
     OPEN INPUT MESSAGE-IN-FILE.
     OPEN OUTPUT MESSAGE-OUT-FILE.
     PERFORM 0400-BUILD-ASSIGN-TABLE.
     PERFORM 0410-GET-MY-SYSTEM-PROCESS.
     PERFORM 0420-GET-MOM-SYSTEM-PROCESS.
```

```
 0200-PROCESS-TRANSACTION.


*   THIS PARAGRAPH PROCESSES ONE MESSAGE FROM $RECEIVE. THE FORMAT OF
*   THE MESSAGE IS PATHSRV-REQUEST. FIRST THE PATHSRV-REPLY MESSAGE IS
*   INITIALIZED. THE SYSTEM AND PROCESS NAME OF THE CURRENT SERVER
*   PROCESS ARE ALWAYS RETURNED TO THE REQUESTER IN THE PATHSRV-REPLY
*   MESSAGE.
*
*   IF THE SUBSIDIARY SERVER CLASS IS SPECIFIED IN THE REQUEST, THEN A
*   PATHSEND SEND IS DONE TO THAT SUBSIDIARY SERVER CLASS.

     MOVE SPACES TO PATHSRV-REPLY.
     MOVE ZERO TO REPLY-CODE OF PATHSRV-REPLY.
     MOVE "N" TO TMF-ABORT-REQUIRED OF PATHSRV-REPLY.
     MOVE WS-PROCESS-NAME OF WS-MY-PROCESS
          TO PROCESS-NAME OF THIS-SERVER OF PATHSRV-REPLY.
     MOVE WS-SYSTEM-NAME OF WS-MY-PROCESS
          TO SYSTEM-NAME OF THIS-SERVER OF PATHSRV-REPLY.


*   IF SERVER-CLASS IN SPECIFIED IN THE REQUEST MESSAGE THEN PROCESSING
*   IS DONE TO PREPARE FOR A PATHSEND SEND TO A SUBSIDIARY SERVER.

     IF SERVER-CLASS OF PATHSRV-REQUEST NOT = SPACES
        MOVE "Y" TO WS-VALID-PATHSRV-REQUEST
        PERFORM 0440-VALIDATE-PATHSRV-REQUEST
        IF WS-VALID-PATHSRV-REQUEST = "Y"


*   IF THE PATHMON-ASSIGN NAME IS SPECIFIED, THEN THE PATHMON SYSTEM
*   AND PROCESS NAME IS LOOKED UP IN THE TABLE OF ASSIGNs. THIS PATHMON
*   SYSTEM AND PROCESS NAME IS USED FOR THE PATHSEND SEND.

            IF PATHMON-ASSIGN-NAME OF PATHSRV-REQUEST NOT = SPACES
                PERFORM 0450-LOOKUP-PATHMON-ASSIGN
            ELSE


*   IF THE PROGRAM LOGIC ARRIVES HERE, THEN AN ASSIGN TO IDENTIFY THE
*   PATHMON IS NOT SPECIFIED. (IF AN EXPLICIT PATHMON SYSTEM AND
*   PROCESS NAME ARE SPECIFIED THEN THEY WILL BE USED FOR THE PATHSEND
*   SEND.)

         IF PATHMON-SYSTEM-AND-PROCESS OF PATHSRV-REQUEST NOT = SPACES
            MOVE PATHMON-SYSTEM-AND-PROCESS OF PATHSRV-REQUEST
                 TO WS-PATHMON-PROCESS-NAME OF WS-SERVERCLASS-SEND-PARAM
         ELSE
```

```
*   IF THE PROGRAM LOGIC ARRIVES HERE, THEN NEITHER AN ASSIGN NOR AN
*   EXPLICIT SYSTEM AND PROCESS NAME ARE USED TO IDENTIFY THE PATHMON
*   TO WHICH THE PATHSEND SEND SHOULD BE DONE. IF THE MOM OF THE
*   CURRENT SERVER PROCESS IS A PATHMON (I.E. WS-SERVER-IS-ASSOCIATIVE
*   = "N") THEN ITS SYSTEM AND PROCESS NAME IS USED FOR THE PATHSEND
*   SEND.

                  IF WS-SERVER-IS-ASSOCIATIVE = "N"
                      MOVE WS-SYSTEM-AND-PROCESS OF WS-MOM-PROCESS
                           TO WS-PATHMON-PROCESS-NAME
                              OF WS-SERVERCLASS-SEND-PARAM
                  ELSE
                    MOVE "PATHMON NOT KNOWN - SERVER IS ASSOCIATIVE" TO
                           NON-SEND-ERROR-MESSAGE OF PATHSRV-REPLY
                    MOVE "N" TO WS-VALID-PATHSRV-REQUEST
                  END-IF
               END-IF
           END-IF
       END-IF

       IF WS-VALID-PATHSRV-REQUEST = "Y"


*   THE REQUEST MESSAGE TO THE SUBSIDIARY SERVER IS ALWAYS SPACES. THE
*   SUBSIDIARY SERVER IS NOT ASKED TO MAKE A REQUEST OF ANOTHER (THIRD
*   LEVEL) SERVER. RATHER IT IS JUST BEING ASKED FOR ITS OWN GUARDIAN
*   PROCESS-ID.

           MOVE SPACES TO SUBSIDIARY-REQUEST
           PERFORM 0460-SEND-TO-SUBSIDIARY-SERVER
           IF WS-SERVERCLASS-SEND-OKAY = "Y"
              MOVE SYSTEM-NAME OF THIS-SERVER OF SUBSIDIARY-REPLY
                  TO SYSTEM-NAME OF SUBSIDIARY-SERVER OF PATHSRV-REPLY
              MOVE PROCESS-NAME OF THIS-SERVER OF SUBSIDIARY-REPLY
                  TO PROCESS-NAME OF SUBSIDIARY-SERVER OF PATHSRV-REPLY
           ELSE
               MOVE "Y" TO TMF-ABORT-REQUIRED OF PATHSRV-REPLY
           END-IF
       ELSE
           MOVE "Y" TO TMF-ABORT-REQUIRED OF PATHSRV-REPLY
       END-IF
    END-IF.


 0300-CLOSEDOWN.
    CLOSE ERROR-LOG-FILE.
    CLOSE MESSAGE-IN-FILE.
    CLOSE MESSAGE-OUT-FILE.
```

```
 0400-BUILD-ASSIGN-TABLE.

*  THIS PARAGRAPH LOADS WS-ASSIGN-TABLE FROM THE STARTUP ASSIGN
*  MESSAGES.
*
*  WS-ASSIGN-TABLE IS USED TO STORE UP TO FIFTY ASSIGN's. THESE
*  ASSIGN's ARE FOR PATHMON SYSTEM AND PROCESS NAMES OF SUBSIDIARY
*  SERVERS. IF PATHMON-ASSIGN-NAME IS USED IN THE REQUEST MESSAGE
*  (PATHSRV-REQUEST) THEN THE ASSOCIATED PATHMON SHOULD BE IN THIS
*  TABLE. THIS IS ONE WAY OF AVOIDING HARD-CODING PATHMON SYSTEM AND
*  PROCESS NAMES.

    PERFORM VARYING WS-MESSAGE-NUMBER OF WS-GETASSIGNTEXT-PARAM
               FROM 1 BY 1
               UNTIL WS-RESULT OF WS-GETASSIGNTEXT-PARAM = -1 OR
                     WS-MESSAGE-NUMBER OF WS-GETASSIGNTEXT-PARAM = 51
       MOVE "LOGICALNAME" TO WS-PORTION OF WS-GETASSIGNTEXT-PARAM
       ENTER "GETASSIGNTEXT"
          USING
             WS-PORTION          OF WS-GETASSIGNTEXT-PARAM
             WS-TEXT             OF WS-GETASSIGNTEXT-PARAM
             WS-MESSAGE-NUMBER   OF WS-GETASSIGNTEXT-PARAM
          GIVING
             WS-RESULT           OF WS-GETASSIGNTEXT-PARAM
       IF WS-RESULT           OF WS-GETASSIGNTEXT-PARAM NOT = -1
          ADD 1 TO WS-NUMBER-OF-ENTRIES OF WS-ASSIGN-TABLE
          MOVE WS-TEXT OF WS-GETASSIGNTEXT-PARAM TO
               WS-ASSIGN-NAME OF WS-ASSIGN-TABLE
                   (WS-NUMBER-OF-ENTRIES OF WS-ASSIGN-TABLE)
       END-IF

       MOVE "TANDEMNAME" TO WS-PORTION OF WS-GETASSIGNTEXT-PARAM
       ENTER "GETASSIGNTEXT"
          USING
             WS-PORTION          OF WS-GETASSIGNTEXT-PARAM
             WS-TEXT             OF WS-GETASSIGNTEXT-PARAM
             WS-MESSAGE-NUMBER   OF WS-GETASSIGNTEXT-PARAM
          GIVING
             WS-RESULT           OF WS-GETASSIGNTEXT-PARAM
       IF WS-RESULT           OF WS-GETASSIGNTEXT-PARAM NOT = -1
          MOVE WS-TEXT OF WS-GETASSIGNTEXT-PARAM TO
             WS-SYSTEM-AND-PROCESS OF WS-ASSIGN-TABLE
                         (WS-NUMBER-OF-ENTRIES OF WS-ASSIGN-TABLE)
       END-IF
    END-PERFORM.
```

```
 0410-GET-MY-SYSTEM-PROCESS.

*  THIS PARAGRAPH GETS THE PROCESS-ID FOR THE CURRENT SERVER PROCESS.
*  THIS PROCESS-ID IS RETURNED TO THE REQUESTER IN PATHSRV-REPLY.

   ENTER TAL "MYPID" GIVING WS-CPU-PIN OF WS-MY-PROCESS.
   ENTER TAL "GETCRTPID" USING  WS-CPU-PIN     OF WS-MY-PROCESS
                                WS-PROCESS-ID  OF WS-MY-PROCESS.
   ENTER TAL "MYSYSTEMNUMBER" GIVING WS-SYSTEM-NUMBER OF WS-MY-PROCESS.
   ENTER TAL "GETSYSTEMNAME"  USING  WS-SYSTEM-NUMBER OF WS-MY-PROCESS
                                     WS-SYSTEM-NAME   OF WS-MY-PROCESS.
 0420-GET-MOM-SYSTEM-PROCESS.

*  THIS PARAGRAPH GETS INFORMATION ABOUT THE MOM OF THE CURRENT SERVER
*  PROCESS. THIS INFORMATION IS USED TO DETERMINE IF THE MOM IS A
*  PATHMON. THE MOM PROCESS IS USED IN PATHSEND SENDS IF THE REQUEST
*  (IN PATHSRV-REQUEST) DOES NOT SPECIFY A PATHMON (EITHER EXPLICITLY
*  OR WITH AN ASSIGN).

    MOVE WS-PROCESS-NAME OF WS-MY-PROCESS
                 TO WS-PROCESS-NAME OF WS-LOOKUPPROCESSNAME-PARAM.
    ENTER TAL "LOOKUPPROCESSNAME" USING WS-LOOKUPPROCESSNAME-PARAM.
    MOVE WS-ANCESTOR-PROCESS-ID OF WS-LOOKUPPROCESSNAME-PARAM
                 TO WS-PROCESS-ID-GENERIC OF WS-MOM-PROCESS.
    IF NOT WS-LOCAL OF WS-MOM-PROCESS
       MOVE ZERO TO WS-SYSTEM-NUMBER OF WS-MOM-PROCESS
       MOVE WS-SYSTEM-NUMBER-1-BYTE OF WS-MOM-PROCESS TO
            WS-BYTE-2 OF WS-SYSTEM-NUMBER-2-BYTES OF WS-MOM-PROCESS
    ELSE
   ENTER TAL "MYSYSTEMNUMBER" GIVING WS-SYSTEM-NUMBER OF WS-MOM-PROCESS
    END-IF.
    ENTER TAL "PROCESSINFO" USING WS-CPU-PIN OF WS-PROCESS-ID-GENERIC
                                             OF WS-MOM-PROCESS
                            WS-PROCESS-ID-GENERIC OF WS-MOM-PROCESS
                              OMITTED
                              OMITTED
                              OMITTED
                              WS-PROGRAM-FILENAME OF WS-MOM-PROCESS
                              OMITTED
                              WS-SYSTEM-NUMBER OF WS-MOM-PROCESS
                              OMITTED
                              OMITTED
                              OMITTED
                              OMITTED
                              OMITTED
                              OMITTED
                              OMITTED
                      GIVING  WS-ERROR OF WS-PROCESSINFO-PARAM.

    IF WS-ERROR OF WS-PROCESSINFO-PARAM NOT = ZERO
       MOVE "Y" TO WS-SERVER-IS-ASSOCIATIVE
    ELSE
       IF WS-FILE OF WS-PROGRAM-FILENAME OF WS-MOM-PROCESS = "PATHMON"
           MOVE "N" TO WS-SERVER-IS-ASSOCIATIVE
       ELSE
           MOVE "Y" TO WS-SERVER-IS-ASSOCIATIVE
       END-IF
    END-IF.

    IF WS-SERVER-IS-ASSOCIATIVE = "N"
       IF WS-LOCAL OF WS-MOM-PROCESS
          MOVE WS-PROCESS-ID OF WS-PROCESS-ID-LOCAL OF WS-MOM-PROCESS
               TO WS-PROCESS-NAME OF WS-MOM-PROCESS
       ELSE
```

```
         STRING "$"
              WS-PROCESS-ID OF WS-PROCESS-ID-NETWORK OF WS-MOM-PROCESS
              DELIMITED BY " "
              INTO WS-PROCESS-NAME OF WS-MOM-PROCESS
      END-IF
    ENTER TAL "GETSYSTEMNAME" USING WS-SYSTEM-NUMBER OF WS-MOM-PROCESS
                                    WS-SYSTEM-NAME OF WS-MOM-PROCESS
    STRING WS-SYSTEM-NAME OF WS-MOM-PROCESS  DELIMITED BY " "
           "."                               DELIMITED BY SIZE
           WS-PROCESS-NAME OF WS-MOM-PROCESS DELIMITED BY " "
       INTO WS-SYSTEM-AND-PROCESS OF WS-MOM-PROCESS
   END-IF.

0440-VALIDATE-PATHSRV-REQUEST.
   IF PATHMON-ASSIGN-NAME OF PATHSRV-REQUEST NOT = SPACES AND
      PATHMON-SYSTEM-AND-PROCESS OF PATHSRV-REQUEST NOT = SPACES
      MOVE "BOTH ASSIGN NAME AND PROCESS NAME SHOULD NOT BE NON-BLANK"
            TO NON-SEND-ERROR-MESSAGE OF PATHSRV-REPLY
       MOVE "N" TO WS-VALID-PATHSRV-REQUEST
   END-IF.
```

```
 0450-LOOKUP-PATHMON-ASSIGN.

*   THIS PARAGRAPH LOOKS UP AN ASSIGN FROM THE REQUEST MESSAGE TO SEE
*   WHETHER A CORRESPONDING  PATHMON SYSTEM AND PROCESS NAME CAN BE
*   FOUND IN THE TABLE OF ASSIGNS BUILT WHEN THE SERVER PROCESS
*   STARTED.  THE RESULTING PATHMON SYSTEM AND PROCESS NAME IS PUT INTO
*   THE PARAMETER VARIABLE THAT IS USED IN THE PATHSEND SEND.

     PERFORM VARYING WS-INDEX OF WS-ASSIGN-TABLE
             FROM 1 BY 1 UNTIL WS-INDEX OF WS-ASSIGN-TABLE >
                               WS-NUMBER-OF-ENTRIES OF WS-ASSIGN-TABLE
    IF WS-ASSIGN-NAME OF WS-ASSIGN-TABLE (WS-INDEX OF WS-ASSIGN-TABLE) =
                             PATHMON-ASSIGN-NAME OF PATHSRV-REQUEST
            MOVE WS-SYSTEM-AND-PROCESS OF WS-ASSIGN-TABLE
                                       (WS-INDEX OF WS-ASSIGN-TABLE)
                 TO WS-PATHMON-PROCESS-NAME OF WS-SERVERCLASS-SEND-PARAM
           MOVE 99 TO WS-INDEX OF WS-ASSIGN-TABLE
        END-IF
     END-PERFORM
     IF WS-INDEX OF WS-ASSIGN-TABLE LESS 99
        MOVE "N" TO WS-VALID-PATHSRV-REQUEST
        STRING "ASSIGN NAME MISSING : "
               PATHMON-ASSIGN-NAME OF PATHSRV-REQUEST
               DELIMITED BY SIZE
               INTO NON-SEND-ERROR-MESSAGE OF PATHSRV-REPLY
     END-IF.

  0460-SEND-TO-SUBSIDIARY-SERVER.

*   THIS PARAGRAPH PERFORMS THE PATHSEND SEND TO THE SUBSIDIARY SERVER

     MOVE "N" TO WS-SERVERCLASS-SEND-OKAY.
     MOVE SERVER-CLASS OF PATHSRV-REQUEST
          TO WS-SERVER-CLASS-NAME OF WS-SERVERCLASS-SEND-PARAM.
     MOVE SUBSIDIARY-REQUEST TO WS-MESSAGE-BUFFER
                                OF WS-SERVERCLASS-SEND-PARAM.

     ENTER TAL "SERVERCLASS_SEND_"

        USING
          WS-PATHMON-PROCESS-NAME          OF WS-SERVERCLASS-SEND-PARAM
          WS-PATHMON-PROCESS-NAME-LEN      OF WS-SERVERCLASS-SEND-PARAM
          WS-SERVER-CLASS-NAME             OF WS-SERVERCLASS-SEND-PARAM
          WS-SERVER-CLASS-NAME-LEN         OF WS-SERVERCLASS-SEND-PARAM
          WS-MESSAGE-BUFFER                OF WS-SERVERCLASS-SEND-PARAM
          WS-REQUEST-LEN                   OF WS-SERVERCLASS-SEND-PARAM
          WS-MAXIMUM-REPLY-LEN             OF WS-SERVERCLASS-SEND-PARAM
          WS-ACTUAL-REPLY-LEN              OF WS-SERVERCLASS-SEND-PARAM
          WS-TIMEOUT                       OF WS-SERVERCLASS-SEND-PARAM
        GIVING
          WS-ERROR                        OF WS-SERVERCLASS-SEND-PARAM.
```

```
*   THE SERVER CLASS OPERATION NUMBER <SCSEND-OP-NUM> PARAMETER IS
*   OMITTED BECAUSE THE SEND IS BEING DONE IN WAIT MODE. IN NOWAIT
*   MODE THE <SCSEND-OP-NUM> IS USED IN CALLS TO AWAITIOX.
*
*   THE <TAG> AND <FLAGS> PARAMETER ARE OMITTED BECAUSE THE SEND IS
*   BEING DONE IN WAIT MODE. THEY ARE ONLY NEEDED IN NOWAIT MODE.

     IF WS-ERROR OF WS-SERVERCLASS-SEND-PARAM = 0
        MOVE "SUCCESSFUL" TO PATHSEND-ERROR OF PATHSRV-REPLY
        MOVE WS-MESSAGE-BUFFER OF WS-SERVERCLASS-SEND-PARAM
             TO SUBSIDIARY-REPLY
        MOVE "Y" TO WS-SERVERCLASS-SEND-OKAY
     ELSE
        IF WS-ERROR OF WS-SERVERCLASS-SEND-PARAM = 233

*  A "233" (SERVER CLASS ERROR) ERROR MUST BE FURTHER ANALYZED WITH
*  SERVERCLASS_SEND_INFO_ TO GET THE PATHSEND ERROR NUMBER AND FILE
*  SYSTEM ERROR NUMBER.

           PERFORM 0500-ANALYZE-SEND-ERROR-233
        ELSE
           MOVE "UNEXPECTED ERROR FOUND AFTER SERVERCLASS_SEND_ CALL"
                TO PATHSEND-ERROR OF PATHSRV-REPLY
        END-IF
     END-IF.


 0500-ANALYZE-SEND-ERROR-233.


*  THIS PARAGRAPH CALLS SERVERCLASS_SEND_INFO_ TO GET THE PATHSEND
*  ERROR NUMBER AND TO GET THE FILE SYSTEM ERROR NUMBER.

     ENTER TAL "SERVERCLASS_SEND_INFO_"

        USING
          WS-PATHSEND-ERROR          OF WS-SERVERCLASS-SEND-INFO-PARAM
          WS-FILE-SYSTEM-ERROR       OF WS-SERVERCLASS-SEND-INFO-PARAM
        GIVING
          WS-ERROR                   OF WS-SERVERCLASS-SEND-INFO-PARAM.

     IF WS-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM NOT = ZERO
        PERFORM 0700-SERVERCLASS-SEND-INFO-ERR
     ELSE
        IF WS-PATHSEND-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM = 907  OR
           WS-PATHSEND-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM = 908  OR
           WS-PATHSEND-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM = 909  OR
           WS-PATHSEND-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM = 910  OR
           WS-PATHSEND-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM = 911  OR
           WS-PATHSEND-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM = 912  OR
           WS-PATHSEND-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM = 919

*           ERROR 907 : INVALID (EXTENDED) SEGMENT ID
*           ERROR 908 : NO (EXTENDED) SEGMENT IN USE
*           ERROR 909 : INVALID VALUE FOR FLAGS PARAMETER
*           ERROR 910 : REQUIRED PARAMETER NOT SUPPLIED
*           ERROR 911 : ONE OF THE BUFFER LENGTH PARAMETERS IS INVALID
*           ERROR 912 : A REFERENCE PARAMETER IS OUT OF BOUNDS
*           ERROR 919 : INVALID VALUE FOR TIMEOUT PARAMETER

            MOVE WS-PATHSEND-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM
                                   TO WS-NUMERIC-DISPLAY
            STRING "BAD PARAMETER PASSED TO SERVERCLASS_SEND_ (ERROR = "
                    WS-NUMERIC-DISPLAY
                   ")"
                DELIMITED BY SIZE
```

```
                INTO PATHSEND-ERROR OF PATHSRV-REPLY
        ELSE
            MOVE WS-PATHSEND-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM
                 TO WS-NUMERIC-DISPLAY
            STRING "SERVERCLASS_SEND_     PATHSEND ERROR NUMBER : "
                   WS-NUMERIC-DISPLAY
                   DELIMITED BY SIZE
                   INTO PATHSEND-ERROR OF PATHSRV-REPLY
        END-IF

        IF WS-FILE-SYSTEM-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM = 48
             MOVE "SERVERCLASS_SEND_ SECURITY VIOLATION (ERROR 48)"
                   TO FILE-SYSTEM-ERROR OF PATHSRV-REPLY
        ELSE
          IF WS-FILE-SYSTEM-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM = 40
              MOVE "SERVERCLASS_SEND_ TIMED OUT (ERROR 40)"
                    TO FILE-SYSTEM-ERROR OF PATHSRV-REPLY
          ELSE
             MOVE WS-FILE-SYSTEM-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM
                  TO WS-NUMERIC-DISPLAY
             STRING "SERVERCLASS_SEND_  FILE SYSTEM ERROR NUMBER : "
                  WS-NUMERIC-DISPLAY
                  DELIMITED BY SIZE
                  INTO FILE-SYSTEM-ERROR OF PATHSRV-REPLY
          END-IF
        END-IF
    END-IF.



0700-SERVERCLASS-SEND-INFO-ERR.

*  THIS PARAGRAPH IS CALLED WHEN AN ERROR IS ENCOUNTERED CALLING
*  SERVERCLASS_SEND_INFO_.

    EVALUATE WS-ERROR OF WS-SERVERCLASS-SEND-INFO-PARAM
        WHEN 2
           MOVE "SERVERCLASS_SEND_INFO_ EXTENDED SEGMENT USAGE ERROR"
                TO NON-SEND-ERROR-MESSAGE OF PATHSRV-REPLY
        WHEN 22
           MOVE "SERVERCLASS_SEND_INFO_ PARAMETER OUT OF BOUNDS"
                TO NON-SEND-ERROR-MESSAGE OF PATHSRV-REPLY
        WHEN 29
           MOVE "SERVERCLASS_SEND_INFO_ MISSING REQUIRED PARAMETER"
                TO NON-SEND-ERROR-MESSAGE OF PATHSRV-REPLY
        WHEN OTHER
           MOVE "UNEXPECTED ERROR FOUND IN CALL TO SERVERCLASS_SEND_INFO_"
                TO NON-SEND-ERROR-MESSAGE OF PATHSRV-REPLY
    END-EVALUATE.


 9000-WRITE-ERROR-LOG-REC.
    WRITE ERROR-LOG-REC.

 9900-STOP-RUN.
    STOP RUN.

;
```

Glossary

Note. This glossary does not include terms for elements of the SCREEN COBOL language that are also found in standard COBOL. For definitions of such terms, refer to standard COBOL texts or to the text of the *Pathway/TS SCREEN COBOL Reference Manual*.

absolute pathname. An OSS pathname that begins with a slash (/) character and is resolved beginning with the root directory. See also OSS pathname, relative pathname, and root directory.

accept operation. An operation in which a screen program waits for a response from the terminal and allows data to be input into the program data area from the terminal.

advisory message. A message displayed in the terminal advisory field to inform the terminal operator of errors detected during input checking.

API. See application program interface (API).

application. A complete set of programs or routines that perform a function. See also Pathway application and NonStop TUXEDO application.

application program interface (API). A set of services (such as programming language functions or procedures) that are called by an application program to communicate with other software components. For example, an API might consist of a set of procedure calls that provide a workstation application with a standard interface for communicating with a Tandem system. Other examples of APIs are the ATMI in BEA TUXEDO systems and NonStop TUXEDO systems and the Pathsend procedures.

application terminal. A terminal on which a Pathway application runs. See also command terminal.

Application-Transaction Monitor Interface (ATMI). The application programming interface to the System/T transaction monitor in a NonStop TUXEDO system. This interface includes transaction routines, message handling routines, service interface routines, and buffer-management routines.

assignment. The use of an ASSIGN command to make logical file assignments for programs in the Guardian environment. A logical assignment equates a Tandem file name with a logical file of a program and, optionally, attributes characteristics to that file.

associative server. A process within a server class that can be started outside the Pathway environment by a process other than the PATHMON process that controls the server class.

ATMI. See Application-Transaction Monitor Interface (ATMI).

attributes. Those characteristics of an object that influence the operation of that object and establish its capabilities.

audited file. A database file that is flagged for auditing by the TMF subsystem; auditing is the monitoring of transactions in preparation for recovery efforts.

audit trail. A record of database changes that can be used by the TMF subsystem to rebuild a database in the event of a hardware or software failure. An audit trail is also known in the industry as a transaction log.

availability. The amount of time an application running on a Tandem system can be used effectively by a user of that application.

backup process. The member of a process pair that takes over the application work when the primary process fails. See also primary process, process pair, and checkpoint message.

base screen. In SCREEN COBOL, a screen that occupies the entire physical display area of a terminal and can be displayed independently of other screens. This type of screen can contain areas on which overlay screens are displayed. See also screen and overlay screen.

batch processing. A method of transaction processing in which transactions are first grouped together and then processed at regular intervals. See also online transaction processing (OLTP).

block mode. A terminal operating mode in which data is read from the terminal and displayed on the terminal one screen at a time. See also conversational mode.

cache. A temporary storage buffer.

cascading server. A term formerly used for a nested server. See nested server.

checkpoint message. In the Guardian environment, a message sent by a primary process to its backup process that keeps the backup process up to date on the state of the application. A checkpoint message provides a snapshot of process activity that can be used in the event of a takeover by a backup process to allow the backup process to maintain fault-tolerant operation.

CISC. See complex instruction-set computing (CISC).

client. An application program that requests services to be performed. In discussions of the Pathway environment, this term is used to refer to the part of an application that runs on some other vendor's hardware, such as a personal computer, Macintosh computer, UNIX workstation, or mainframe computer system, and makes requests of a server process. See also requester, server, and client/server model.

client/server model. A model for distributing applications. In general, but not always, in this model the client process resides on a workstation and the server process resides on a second workstation, minicomputer, or mainframe system. Communication takes the form of request and reply pairs, which are initiated by the client and serviced by the server. (A server can make requests of another server, thus acting as a client.) Client/server computing is often used to connect different types of workstations or

personal computers to a host computer system by means of supported communications protocols. See also requester/server model.

client/transaction server model. A model for client/server applications. The client/transaction server model is the model of choice for high-volume OLTP applications in which transaction volume is great and the processing requirements change infrequently.

In the Tandem environment, an application following this model divides processing between a client running on a workstation and servers running on a Tandem system. The client handles the user interface and business logic and processing. The servers store information for use by the client and handle database input and output functions. Interprocess communication (IPC) messages transfer data between client and server.

COBOL85. The Tandem compiler and run-time support for the American National Standards Institute (ANSI) programming language COBOL, X.3.23-1985. Pathway server processes are often written in this language.

cold start. The operation that starts a PATHMON environment for the first time. This operation either creates a new PATHMON configuration file (PATHCTL file) that defines the PATHMON environment and its objects or overwrites an existing PATHMON configuration file (which effectively creates a new PATHMON environment). See also cool start.

command file. A file that serves as a source for command input. For example, users can prepare a command file containing PATHCOM or SCREEN COBOL Utility Program (SCUP) commands. They can then cause the commands in the file to be executed by issuing the PATHCOM or SCUP OBEY command and specifying the name of the file. Alternatively, they can specify this file as the input file when they execute PATHCOM or SCUP.

command interpreter. An interactive program used to run programs, check system status, create and delete disk files, and alter hardware states.

command terminal. A terminal at which a system manager or operator enters commands for configuration and management, such as the PATHCOM commands that configure and manage a PATHMON environment. See also application terminal.

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. See also reduced instruction-set computing (RISC).

configuration. The definition or alteration of characteristics of an object. See also object.

configured TERM object. A TERM object that is explicitly configured with an ADD TERM command. Such a TERM object exists until it is explicitly deleted. Names of configured TERM objects begin with a letter. See also temporary TERM object and TERM object.

consistency. See database consistency.

context. Information required by a server to process the current request in an exchange of multiple request and reply messages: for example, identification of the last item processed. See also context-free server and terminal context.

context-free server. A server that does not retain any information about the processing of previous requests. A context-free server accepts a single message from a requester, performs the requested tasks, and issues a single reply to respond to the requester. After the reply message is issued, the server retains no information, or context, that can be used in subsequent requests. In general, context-free servers are relatively simple to program and can be restarted quickly, but they require the requester to pass context information to the server on each request. Servers handling requests from Pathsend requesters can be either context-free or context-sensitive, but servers servicing requests from SCREEN COBOL requesters must be context-free. A context-free server is analogous to a NonStop TUXEDO request/response server. Tandem subsystems are context-free servers; therefore, management applications using the Subsystem Programmatic Interface (SPI) to communicate with Tandem subsystems must pass back context information in continuation requests. See also context, context-sensitive server, and Subsystem Programmatic Interface (SPI).

context-sensitive server. A server that retains information about the processing of previous requests. A context-sensitive Pathway server can engage in a multiple-message communication, or dialog, with a requester. Because context-sensitive servers must maintain message context for the dialog, they are more complex to program than context-free servers. They typically have longer restart times because they must recover the requester context. See also context and context-free server.

context sensitivity. The ability of a requester to exchange a series of multiple request and reply messages (that is, a dialog) with a particular server process. See also context-sensitive server and dialog.

conversation. See dialog.

conversational mode. (1) A terminal operating mode in which data is read from the terminal and displayed on the terminal screen one line at a time. See also block mode and intelligent mode. (2) The mode of communication that enables an ongoing dialog between a client (or requester) and a server. Data is sent and received in an iterating fashion without return to the transaction monitor until the application dialog is completed. Multiple messages can be exchanged between the client and server participating in the communication. See also conversational server.

conversational model. A model for requester-server communication that enables an ongoing dialog between a client (or requester) and a server. Multiple messages can be exchanged between the client and server process before control is returned to the transaction monitor. See also request/response model and conversational server.

conversational server. A server that offers conversational services and can participate in a conversation, or dialog, with a client; that is, a context-sensitive server. See also conversational mode (definition 2), request/response server, and context-sensitive server.

cool start. The operation that restarts a PATHMON environment, using the information in an existing PATHMON configuration file (PATHCTL file). The PATHMON environment must have been previously started with a cold start operation. See also cold start.

Crossref cross-reference generator. A Tandem software tool that produces a cross-referenced listing of selected identifiers—such as data variables, statement labels, or subprograms—in an application program.

current working directory. The OSS directory from which relative pathnames are resolved. See also OSS pathname and relative pathname.

database consistency. The state of a database in which items satisfy established criteria. For example, an account balance must equal credits to the balance minus debits to the balance. When the database satisfies these criteria, the database is considered to be consistent. In general, a database is consistent when it is accurate and all changes generated by transactions are complete. Database consistency is defined by the application, which establishes the values and relationships of database fields and records.

database management system (DBMS). A product, such as NonStop SQL/MP or Enscribe, that serves as the interface between a user or program (for example, a Pathway server) and the database. Among its many functions, the DBMS controls access to and organization of data within the database.

Data Definition Language (DDL). (1) The set of data definition statements within the Structured Query Language (SQL). (2) A Tandem product for defining data objects in Enscribe files and translating object definitions into source code.

data integrity. The condition of a database when its data values are accurate, valid, and consistent according to rules established for changing the database. See also database consistency.

DBCS. See double-byte character set (DBCS).

DDL. See Data Definition Language (DDL).

deadlock. (1) A situation in which two processes cannot proceed because each is waiting for a reply from the other. (2) A situation in which two transactions cannot proceed because each is waiting for the other to release a lock.

dedicated device. A term formerly used for a terminal or other input/output device controlled by a configured TERM object, so that a Pathway application always ran on that device without having to be started from PATHCOM with a RUN PROGRAM command. (No new term replaces this term; instead, the manual text now refers to such devices as those associated with configured TERM objects.) See also nondedicated device and configured TERM object.

default value. The value that the system uses for a particular attribute or parameter when a value has not been supplied by the user.

DEFINE. A named set of attributes and associated values. In a DEFINE (as with an ASSIGN command), users can specify information to be communicated to processes they start.

definition files. A set of files containing data declarations for items related to SPI messages and their processing. The core definitions required to use SPI are provided in a DDL file and in several language-specific definition files, one for each programming language that supports SPI. The Tandem DDL compiler generates the language-specific files from the DDL file. Subsystems that support SPI provide additional definition files containing subsystem-specific definitions.

delimiters. Characters that make it possible for a SCREEN COBOL requester and an external device or front-end process to exchange compact variable-length messages efficiently; delimiters can be message delimiters or field delimiters.

descriptor. For each elementary data item, the SCREEN COBOL compiler builds a data structure that describes the size, type, usage, and dependencies of the item. All of the information that pertains to a given item makes up the descriptor for that item. For example, the PICTURE specification is included in the descriptor. The descriptors are passed to the TCP in the pseudocode and provide a dictionary of information for interpreting and handling incoming data. When the MAP or SMAP compiler option is used, the descriptors appear in the compiler map at the end of the listing.

diagnostic screen. A screen of information that is displayed to inform the terminal operator of error conditions and termination status.

dialog. A multiple-message communication between a requester and a context-sensitive server. A dialog is also called a conversation. See also context sensitivity and context-sensitive server.

disk process. In the Tandem NonStop Kernel operating environment, the portion of the operating-system software that performs read, write, and lock operations on disk volumes and creates TMF audit records. See also file system.

display attribute. A terminal display feature that is given a screen data name. The screen data name can be associated with a predefined system name in the SPECIAL-NAMES paragraph and thus be manipulated by a SCREEN COBOL program.

distributed data. Information (for example, customer names and addresses, inventory items, and personnel records) that resides on more than one node in a network and can be accessed by authorized users from any node in that network.

distributed processing. A type of processing environment in which resources are distributed among CPUs within a single system or spread across a network of systems. A user on any network node can, if properly authorized, access resources and database files anywhere within the network.

Distributed Systems Management (DSM). A group of tools for managing a variety of subsystems in a distributed processing environment.

distributed transaction processing (DTP). The coordination of transactions among application servers residing within an Expand network and possibly accessing different database management systems (NonStop SQL/MP and Enscribe). DTP allows the coordination of multiple, autonomous actions as a single logical unit of work.

double-byte character. A character represented in two bytes. See also double-byte character set.

double-byte character set (DBCS). A character set, such as Tandem Kanji, that uses two bytes of data to represent a single character.

DSM. See Distributed Systems Management (DSM).

dumb terminal. See fixed-function terminal.

dynamic server. A server process that the PATHMON process creates after a TCP or LINKMON process has waited for a specified time period for a static server to become available. A dynamic server process exists only as long as it is needed. See also static server.

EDIT file. A source text file that can be augmented and modified by the user through a Tandem text editor program such as TEDIT (PS Text Edit).

EMS. See Event Management Service (EMS).

EMS log file. See event log file.

Enable product. A tool provided by Tandem that allows users to develop simple data management applications without using a conventional programming language. The Enable product can generate SCREEN COBOL programs for use with Enscribe databases.

Enscribe database record manager. Tandem database management software that provides a record-at-a-time interface between servers and a distributed database. See also NonStop SQL/MP.

event log file. A file maintained by the Event Management Service (EMS) for logging of event messages.

Event Management Service (EMS). A part of DSM used to provide event collection, event logging, and event distribution facilities. It provides for different descriptions of events for people and for programs, lets an operator or application select specific event-message data, and allows for flexible distribution of event messages within a system or network. EMS has an SPI-based programmatic interface for reporting and retrieving events. See also event message.

event message. A special kind of SPI message that describes an event occurring in the system or network. Event messages are collected, logged, and distributed by EMS. See also Event Management Service (EMS).

expandability. See scalability.

Expand networking software. Tandem software that can connect up to 255 Tandem NonStop systems into a single network.

Extended General Device Support (GDSX). A Tandem product that facilitates communication between general I/O devices and a PATHMON environment by acting as a front-end or a back-end process.

extensible structured token. In the Subsystem Programmatic Interface (SPI), a token with a value that can be extended by appending new fields in later releases. The token is accessed through reference to a token map containing field-version and null-value information, allowing SPI to provide compatibility between different versions of the structure. See also simple token and token (definition 2).

external PATHMON process. See external process.

external process. A process in a different PATHMON environment from the process with which it is communicating. For example, suppose a TCP managed by PATHMON process $PMB requests a link to a server process in a server class that is managed by PATHMON process $PMA. Both the TCP and PATHMON process $PMB are external processes with respect to PATHMON process $PMA and the server class managed by $PMA.

external server. See external process.

external TCP. See external process.

fault tolerance. The ability of a Tandem NonStop system to continue processing despite the failure of any single software or hardware component within the system.

field-characteristic clause. In SCREEN COBOL, an ordered set of characters that specify the characteristics of a screen field.

file identifier (file ID). In the Guardian environment, the portion of a file name following the subvolume name. In the OSS environment, a portion of the internal information used to identify a file in the OSS file system. The two identifiers are not comparable.

file name. In the Guardian environment, the set of node name, volume name, subvolume name, and and file identifier characters that uniquely identifies a file. This name is used to open a file and thereby provide a connection between the opening process and the file. See also fully qualified file name, partially qualified file name, OSS filename, and OSS pathname.

file-name expansion. The expansion of a partially qualified Guardian file name for a disk file to include the associated node, volume, and subvolume names.

file system. (1) In the Guardian environment, the application program interface for communication between a process and a file. A file can be a disk file, a device other than a disk, or another process. (2) In the OSS environment, a collection of files and file

attributes. A file system provides the namespace for the file serial numbers that uniquely identify its files.

File Utility Program (FUP). A Tandem product that allows users to create, copy, purge, and otherwise manipulate disk files interactively.

fixed-function terminal. A nonintelligent device (that is, a device without processing ability) capable of sending and receiving information over communications lines. Fixed-function terminals are often referred to as dumb terminals.

freeze condition. A condition in which communication between a terminal and a server class is prohibited. See also thaw condition.

front-end process. A process, such as the process that accepts the TCP terminal-data stream, that serves as the intermediary between one system, process, or device and another.

fully qualified file name. The complete name of a file in the Guardian environment. For a permanent disk file, this consists of a node name (system name), volume name, subvolume name, and file identifier (file ID). In interactive interfaces such as PATHCOM and TACL, the parts of a file name are separated by periods. See also partially qualified file name.

gateway process. A process, such as the Transaction Delivery Process (TDP) that is part of RSC, that manages communications between dissimilar environments (for example, a workstation and a Tandem system). A gateway process both transfers information and converts it to a form compatible with the protocols used by the destination environment.

GDSX. See Extended General Device Support (GDSX).

graphical user interface (GUI). A type of screen interface that typically includes pull-down menus, icons, dialog boxes, and online help.

Guardian. An environment available for interactive or programmatic use with the Tandem NonStop Kernel. Processes that run in the Guardian environment use the Guardian system procedure calls as their application program interface, and might also use related APIs such as the Pathsend and TMF procedure calls. See also Open System Services (OSS).

Guardian environment. The Guardian API, tools, and utilities. See also Guardian.

Guardian operating environment. See Guardian environment.

high PIN. A process identification number (PIN) in the range 256 through 65535. See also process identification number (PIN) and low PIN.

IDS. See intelligent device support (IDS) facility.

Inspect. The Tandem debugging tool that can be used interactively to examine and modify the execution of Guardian processes and SCREEN COBOL requester programs.

Inspect command terminal. The terminal on which programmers enter commands to Inspect when debugging a SCREEN COBOL program or a Pathway server.

intelligent device. A device such as an automatic teller machine, a point-of-sale device, or a communications line, or a process such as a Guardian process, that can communicate with the Pathway environment through the intelligent device support (IDS) facility, the Remote Server Call (RSC) product, or the Pathsend procedure calls.

intelligent device support (IDS) facility. A feature of the TCP that supports access to Pathway server classes by intelligent devices. This facility allows SCREEN COBOL requester programs to interact with external processes that, in turn, control devices such as personal computers, automated teller machines, and point-of-sale devices.

intelligent mode. An operating mode in which data and messages are sent between an intelligent device and the Pathway environment. See also conversational mode (definition 1), intelligent device, and message-oriented requester.

interactive mode. An operating mode in which commands are entered from a terminal keyboard.

interoperability. The ability to communicate, execute programs, or transfer data between dissimilar environments.

interoperate. To communicate, execute programs, or transfer data between dissimilar environments.

interprocess communication (IPC) message. The unit of communication between requesters and servers. An IPC message consists of a request message and a reply message.

I/O process. In the Guardian environment, a system process to manage input/output devices. Applications use the Guardian file system to send requests to I/O processes. See also file system.

IPC message. See interprocess communication (IPC) message.

keyword. A word in a command string or programming language that must be spelled and positioned in a prescribed way, usually to indicate the meaning of an adjacent parameter.

library. A set of related files or common files that can be accessed by multiple programs or processes.

linear expandability. See scalability.

link. (1) An open of a server process within a server class. When a link manager—that is, a TCP or a LINKMON process—sends a request to a PATHMON process for a link to a server in a specified server class, the PATHMON process selects a server process in that server class (possibly starting a new server process if necessary) and then returns the name of the server process to the requesting link manager. See also link granting and link manager. (2) To examine, collect, associate together, and modify code and data

blocks from one or more object files to produce a target object file. On Tandem NonStop systems, linking for TNS/R native object files is performed by the `nld` utility.

link access. The actual transfer of data from a requester to a server process. In the Pathway environment, link access is provided by TCPs and LINKMON processes. See also link granting.

link granting. The process of selecting a particular server process in a server class to handle a request from a link manager (TCP or LINKMON process) on behalf of a requester. In the Pathway environment, link granting is done by the PATHMON process. See also link and link access.

link management. The act of coordinating the sharing of links between requester or client processes and server processes.

link manager. A process that requests links to server processes and provides link access after the link is granted. TCPs and LINKMON processes are the link managers in the Pathway environment. See also link access and link granting.

LINKMON process. A Guardian process that supports access to servers in the Pathway and NonStop TUXEDO environments. LINKMON processes act as link managers for requesters and clients that use the Pathsend procedure calls and other interfaces such as RSC, POET, and the NonStop TUXEDO ATMI. See also link manager.

lock. A mechanism that coordinates access to the same data; locks are either shared or exclusive.

log file. See PATHMON log file and event log file.

low PIN. A process identification number (PIN) in the range 0 through 255. (PIN 255 is reserved by the system; it is never assigned to a running process, but is used by high-PIN processes to communicate with low-PIN processes.) See also process identification number (PIN) and high PIN.

manageability. The ability to easily and comprehensively manage a subsystem, system, or network.

MAKEUL. A TACL macro used to perform pTAL compilation of user-written conversion routines for use with the Pathway/TS TCP and SCREEN COBOL requesters and to create the TNS/R native TCP user library containing these routines.

management application. An application program that automates configuration and management tasks. Such a program can request from the PATHMON process the same kinds of services that system managers can request through the PATHCOM interface. A management application can also interact with subsystems other than the Pathway subsystem. Management applications use the Subsystem Programmatic Interface (SPI) to send commands to subsystems and the Event Management Service (EMS) to receive notification of significant events.

message-oriented requester. A SCREEN COBOL requester that sends data from working storage to a device (or to a front-end process that controls a device) and receives data from the device or process into working storage by way of Message Section templates. SCREEN COBOL requesters that use the intelligent device support (IDS) facility are message-oriented. These requesters use SEND MESSAGE statements and their REPLY clauses in the Procedure Division to interact with the intelligent devices or front-end processes. See also screen-oriented requester.

Message Section. A section in the Data Division of a SCREEN COBOL source program that describes the data type, size, and relative position (sequence) of each field in a message template. This section also defines the editing and conversion that must be performed on each field. See also message-oriented requester.

mixed data item. A data item that contains both single-byte and double-byte characters; in a COBOL or SCREEN COBOL program, these data items are declared as PIC X.

modified data tag (MDT). In SCREEN COBOL, a bit that is set or reset to indicate whether data in an associated field is to be sent to the computer from the terminal.

multithreaded. A programming model that provides more than one thread of control within a program. Multithreading allows multiple sequential processing tasks to be executed concurrently within a process: for example, a terminal control process (TCP). See also thread and single-threaded.

native System/T client. In the NonStop TUXEDO environment, a client program that executes in the Open System Services (OSS) environment and communicates directly with System/T. An example of this type of client is the Pathway translation server for the NonStop TUXEDO system. Also called a NonStop TUXEDO native client.

nested server. A server that acts as a requester by sending requests to other servers. In the Pathway environment, such requests are made by calls to Pathsend procedures.

nld utility. A utility that collects, links, and modifies code and data blocks from one or more object files to produce a target TNS/R native object file.

node. A Tandem NonStop system that is part of an Expand network. The name of the node, also called the system name, is the first of four parts of a file name in the Guardian environment. See also Expand networking software.

no-early-reply rule. The rule that states that when a server process reads a request message, it should completely process the request before it replies to it.

nondedicated device. A term formerly used for a terminal or other input/output device on which a Pathway application could be started from PATHCOM with a RUN PROGRAM command. The RUN PROGRAM command results in the creation of a temporary TERM object to control the terminal. (No new term replaces this term; instead, the manual text now refers to such devices as those associated with temporary TERM objects.) See also dedicated device and temporary TERM object.

noninteractive mode. An operating mode in which commands are entered through a command file.

NonStop Kernel. See Tandem NonStop Kernel.

NonStop processing. On Tandem NonStop systems, processing characterized by continued operation even when a component fails, when equipment is being repaired or replaced, or while new processors or peripheral devices are being added to the system. In the Guardian environment, NonStop processing is provided by means of fault tolerance and process pairs.

NonStop SQL/MP. The Tandem relational database management system that promotes efficient online access to large distributed databases. See also Structured Query Language (SQL) and Enscribe database record manager.

NonStop Transaction Manager/MP (NonStop TM/MP). A Tandem software product that provides transaction management, transaction protection, and database consistency in online transaction processing (OLTP) environments. It gives full protection to transactions that access NonStop SQL/MP and Enscribe databases, as well as recovery capabilities for transactions, online disk volumes, and entire databases. The component of NonStop TM/MP that provides these features is the TMF subsystem. See also Transaction Management Facility (TMF) subsystem.

NonStop Transaction Services/MP (NonStop TS/MP). A Tandem product that provides process management and link management functions for OLTP applications on Tandem NonStop systems. NonStop TS/MP consists of the PATHMON process, the LINKMON process, the PATHCOM process and interface, and the Pathsend procedures. Together with NonStop Transaction Manager/MP (NonStop TM/MP), NonStop TS/MP forms the foundation for Tandem's open transaction processing services, including those provided by the RSC and POET products and the NonStop TUXEDO system. See also Pathway/TS and NonStop Transaction Manager/MP (NonStop TM/MP).

NonStop TUXEDO application. The collection of machines, servers and services, and System/T components defined by a single configuration file in the NonStop TUXEDO environment. See also NonStop TUXEDO transaction processing environment and Pathway application.

NonStop TUXEDO environment. See NonStop TUXEDO transaction processing environment.

NonStop TUXEDO native client. See native System/T client.

NonStop TUXEDO server. A server process or program managed by the NonStop TUXEDO system administrative facilities. See also Pathway server.

NonStop TUXEDO system. Tandem's implementation of the BEA Systems, Inc., TUXEDO enterprise transaction processing system. Major features of the NonStop TUXEDO implementation include the use of the Tandem core services (Tandem NonStop Kernel, NonStop Transaction Services/MP, and NonStop Transaction Manager/MP) to provide the Tandem fundamentals (scalability, availability, and manageability) for TUXEDO

applications. The NonStop TUXEDO system runs in the OSS operating environment on Tandem NonStop systems. See also OSS environment.

NonStop TUXEDO transaction processing environment. An environment that provides the API and transaction monitor functions of the BEA Systems, Inc., TUXEDO transaction processing system in addition to the benefits provided by the Tandem core services: Tandem NonStop Kernel, NonStop Transaction Services/MP, and NonStop Transaction Manager/MP. See also Pathway transaction processing environment.

OBEY command. A command in a Tandem interactive interface, such as PATHCOM or the SCREEN COBOL Utility Program (SCUP), that allows users to execute the commands in a command file. See also command file.

object. An entity that is subject to independent reference or control by one or more subsystems. Examples of objects are devices, communications lines, processes, and files. In the PATHMON environment, the types of objects referred to or controlled by PATHCOM are PATHMON, PATHWAY, LINKMON, SERVER, TCP, TERM, PROGRAM, and TELL. The PATHWAY object, used with the SET PATHWAY and STATUS PATHWAY commands, refers to an entire PATHMON environment. The LINKMON object is only referred to, not controlled, by PATHCOM: that is, users can use PATHCOM to obtain information about LINKMON processes but cannot use it to make any changes in the configuration or state of those processes. See also PATHMON environment.

object attributes. See attributes.

object file. A file generated by a compiler, linker, or binder that contains machine instructions and other information needed to construct the executable code spaces and initial data for a process. The file can be a complete program that is ready for immediate execution, or it can be incomplete and require linking with other object files before execution. Compilers for languages such as COBOL85 produce object code. See also pseudocode file.

object type. In the Subsystem Programmatic Interface (SPI) or in an interactive interface such as PATHCOM, a category of objects to which a specific object belongs. Object types for PATHCOM include PATHMON, PATHWAY, LINKMON, SERVER, TCP, TERM, PROGRAM, and TELL. The SPI interface to the Pathway subsystem uses different names for some of these object types and defines additional object types; for example, PM, LM, PROGTERM, and TCPTERM are object types in the SPI interface.

OLTP. See online transaction processing (OLTP).

OLTP application. See online transaction processing (OLTP) application.

online transaction processing (OLTP). A method of processing transactions in which entered transactions are immediately applied to the database. The information within the database is readily available to all users through online screens and printed reports. The transactions are processed while the requester waits, as opposed to queued or batched transactions, which are processed at a later time. Online transaction processing can be

used for many different kinds of business tasks such as order processing, inventory control, accounting functions, and banking operations. See also batch processing.

online transaction processing (OLTP) application. A set of programs that perform online transaction processing (OLTP) tasks on behalf of the user. With an OLTP application, many terminal users can update data simultaneously, recording the changes in the database as they are entered. OLTP applications generally display, check, and accept input data; manipulate the input data; and perform some type of data-output activity.

Open System Services (OSS). An open system environment available for interactive or programmatic use with the Tandem NonStop Kernel. Processes that run in the OSS environment use the OSS application program interface; interactive users of the OSS environment use the OSS shell for their command interpreter. See also Guardian and Guardian environment.

OSS. See Open System Services (OSS).

OSS environment. The NonStop Kernel Open System Services (OSS) API, tools, and utilities. See also Open System Services (OSS).

OSS filename. A component of an OSS pathname containing any valid characters other than a slash (/) or a null. See also file name, OSS pathname, and file system (definition 2).

OSS operating environment. See OSS environment.

OSS pathname. The string of characters that uniquely identifies a file within its file system in the OSS environment. A pathname can be either absolute or relative. See also absolute pathname, relative pathname, and file system (definition 2).

OSS username. A string that uniquely identifies a user within the user database for a node.

overlay area. In SCREEN COBOL, an area of a base screen within which an overlay screen can be displayed.

overlay screen. In SCREEN COBOL, a screen that is displayed in an overlay area of a base screen. A base screen can be used with various overlay screens. See also screen and base screen.

partially qualified file name. A Guardian file name in which only the right-hand file-name parts are specified. The remaining parts of the file name assume default values. See also fully qualified file name.

PATHCOM. (1) The interactive interface to the PATHMON process, through which users enter commands to configure and manage Pathway applications. (2) The process that provides this interface.

PATHCOM command file. A file of PATHCOM commands that define and add the PATHMON-controlled objects required to execute an application. This file can contain all of the commands needed to start a PATHMON environment.

PATHCOM command terminal. See command terminal.

PATHCTL. See PATHMON configuration file.

Pathmaker product. A menu-driven application generator, provided by Tandem, that increases the productivity of programmers developing Pathway applications. The Pathmaker software generates requester programs in SCREEN COBOL and server programs in C or COBOL85.

PATHMON configuration file. A disk file in which a PATHMON process maintains configuration information for the objects under its control. The name of this file is PATHCTL.

PATHMON-controlled object. An object defined and managed by a PATHMON process, through PATHCOM or the Pathway management programming interface. In the PATHCOM interface, such an object can be of type PATHWAY, PATHMON, SERVER, TCP, TERM, PROGRAM, or TELL. See also object and Pathway object.

PATHMON environment. The servers, server classes, TCPs, terminals, SCREEN COBOL programs, and tell messages that run together under the control of one PATHMON process.

PATHMON log file. A file used by a PATHMON process for reporting errors and changes in status.

PATHMON object. An object of type PATHMON; that is, a PATHMON process. See also PATHMON process and PATHMON-controlled object.

PATHMON process. The central controlling process in the Pathway environment. The PATHMON process maintains configuration-related data; grants links to server classes in response to requests from TCPs and LINKMON processes; and performs all process control (starting, monitoring, restarting, and stopping) of server processes and TCPs.

pathname. See OSS pathname.

Pathsend procedures. The set of Guardian procedure calls that provide general access to Pathway server classes from any process on a Tandem system.

Pathsend process. A process, written as a Guardian program in C, C++, COBOL85, Pascal, pTAL, or TAL, that makes calls to Pathsend procedures to request services from a Pathway server. A Pathsend process can be either a standard requester, which initiates application requests, or a nested server, which is configured as a server class but acts as a requester by making requests to other servers. A Pathsend process is also known as a Pathsend requester.

Pathsend program. A Guardian program, written in C, C++, COBOL85, Pascal, pTAL, or TAL, that makes calls to Pathsend procedures to request services from a Pathway server. A running Pathsend program is called a Pathsend process. See also Pathsend process.

Pathsend requester. See Pathsend process.

PATHTCP2. The TCP object file, usually identified by the file name $SYSTEM.SYSTEM.PATHTCP2.

PATHTCPL. The TCP user library object file.

Pathway application. A set of programs that perform online transaction processing tasks in the Guardian environment, using interfaces defined by Tandem. A Pathway application can include SCREEN COBOL requesters, Pathsend requesters, and Pathway servers running on Tandem NonStop systems. It can also include GDSX front-end processes and clients that use RSC or POET. See also NonStop TUXEDO application.

Pathway application development environment. A set of tools supporting the development of applications for the Pathway transaction processing environment. Depending on the customer's needs and software configuration, this set of tools could include the SCREEN COBOL compiler and the SCREEN COBOL Utility Program (SCUP), the POET application development tools, the Pathmaker product, and various tools from Tandem Alliance partners. See also Pathway transaction processing environment.

Pathway environment. See Pathway transaction processing environment.

Pathway management programming interface. A set of programmatic commands that allow users to write management application programs that communicate directly with the PATHMON process for configuration and management. This interface is based on the Subsystem Programmatic Interface (SPI) within the Distributed Systems Management (DSM) software. Programmatic commands communicating with the PATHMON process use the Pathway subsystem ID. See also Pathway subsystem and subsystem ID.

Pathway monitor process. See PATHMON process.

Pathway object. An object in the Pathway transaction processing environment. The set of Pathway objects is a more inclusive set than the set of PATHMON-controlled objects: for example, a LINKMON process is a Pathway object but not a PATHMON-controlled object. See also object, PATHMON-controlled object, and Pathway transaction processing environment.

Pathway Open Environment Toolkit (POET). A set of programs and utilities that helps programmers create and run client/transaction server applications. In the POET environment, the client is a program running on a workstation, and the server is a Pathway server running on a Tandem NonStop system. POET uses the services of the Remote Server Call (RSC) product. The programming tools provided by POET include a simplified programming interface, name mapping, and conversion mapping.

Pathway server. A server process or program in the Pathway transaction processing environment. See also NonStop TUXEDO server.

Pathway subsystem. The PATHMON environment components to which SPI commands are sent under the Pathway subsystem ID and which generate EMS event messages with the Pathway subsystem ID. All SPI commands for the Pathway subsystem are sent to the PATHMON process, but the processsing for the command might involve other processes, such as a TCP. Likewise, all EMS messages from the Pathway subsystem are generated by the PATHMON process, but the information might originate from another process. See also subsystem ID.

Pathway system. A term formerly used for the set of objects managed by a particular PATHMON process; now called PATHMON environment. See PATHMON environment.

Pathway transaction processing environment. A run-time environment consisting of Tandem's transaction-processing products for the Guardian operating environment. This term is often shortened to "Pathway environment." Depending on the customer's needs and software configuration, the Pathway environment could include NonStop TS/MP, the run-time portions of Pathway/TS (the TCP and the SCREEN COBOL run-time environment), NonStop TM/MP, GDSX processes, the run-time portion of the RSC product, the POET run-time environment, and the TRANSFER delivery system (when used as a workflow aid in transaction processing). See also NonStop TUXEDO transaction processing environment.

Pathway translation server for the NonStop TUXEDO system. A server process, provided by Tandem as part of the NonStop TUXEDO product, that allows a Pathway (SCREEN COBOL or Pathsend) requester to use the services of a NonStop TUXEDO server. The translation server thus acts as a gateway process between the Pathway environment and the NonStop TUXEDO environment. Requesters that use this translation server must include special information in the header of each request message to identify the target NonStop TUXEDO application and service.

Pathway/TS. A Tandem product that provides tools for developing and interpreting screen programs to support OLTP applications in the Guardian environment on Tandem NonStop systems. Pathway/TS screen programs communicate with terminals and intelligent devices. Pathway/TS includes the TCP, the SCREEN COBOL compiler and run-time environment, and the SCREEN COBOL Utility Program (SCUP). It requires the services of the NonStop TS/MP product. See also NonStop Transaction Services/MP (NonStop TS/MP).

PIN. See process identification number (PIN).

POBJ. The default prefix, or file-name root, used by the SCREEN COBOL compiler in naming its output files. If no prefix is specified in the RUN command to run the compiler, the compiler produces a code file named POBJCOD, a directory file named POBJDIR, and (if the SYMBOLS option is enabled) a symbols file named POBJSYM.

POET. See Pathway Open Environment Toolkit (POET).

Portable Transaction Application Language (pTAL). A machine-independent systems programming language based on TAL. The pTAL language excludes architecture-specific TAL constructs and includes new constructs that replace the architecture-specific constructs. See also Transaction Application Language (TAL).

primary process. The currently active process of a process pair in the Guardian environment. See also backup process and process pair.

process. (1) A unique execution of a program in the Guardian environment. (2) An entity in the OSS environment consisting of an address space, a single thread of control that executes within that address space, and the system resources required by that thread of control. See also process type.

process identification number (PIN). An unsigned integer that identifies a process within a processor module in a Tandem NonStop system.

process management. The act of configuring, creating, and initializing processes; the monitoring and stopping of processes; and the recovery of failed processes. The PATHMON process provides process management functions for OLTP applications on Tandem NonStop systems.

process pair. A fault-tolerant arrangement of processes in the Guardian environment, whereby two processes in separate processors share the same name and execute identical code. One process functions as the primary process and the other functions as the backup process. The two processes are kept in sync through checkpoint messages sent from the primary to the backup process. If the primary process fails, the backup process is notified that it is now the primary, and it resumes the application work from the last valid checkpoint message.

process type. An attribute of a server class indicating whether the server processes in that server class are Guardian processes or OSS processes. See also process.

program file. An executable object file. See also object file.

PROGRAM object. A template for creating and starting temporary TERM objects. See also temporary TERM object.

pseudocode file. A file containing compiled code that is interpreted by software instead of being executed by the hardware. The SCREEN COBOL compiler produces pseudocode to be interpreted by the TCP. See also object file.

pseudo Pathsend procedure. In the Extended General Device Support (GDSX) software, one of the TSCODE-supported procedures that have Pathsend procedure counterparts.

pTAL. See Portable Transaction Application Language (pTAL).

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. See also complex instruction-set computing (CISC).

relative pathname. An OSS pathname that does not begin with a slash (/) character. A relative pathname is resolved beginning with the current working directory. See also OSS pathname, absolute pathname, and current working directory.

Remote Duplicate Database Facility (RDF). The Tandem software product that assists in disaster recovery for OLTP production databases, monitors database updates audited by the TMF subsystem on a primary system, and applies those updates to a copy of the database on a remote system.

Remote Server Call (RSC). A Tandem software product that facilitates client/server computing, allowing personal computer (PC) or workstation applications running under Microsoft Windows software or the MS-DOS or OS/2 operating system to access Pathway server classes and Guardian processes. Transactions are transmitted from the PC or workstation application (the client) to a Pathway application running on a Tandem NonStop system (the server) by means of a supported communications protocol, such as NETBIOS, TCP/IP, or an asynchronous connection.

reply message. The part of an interprocess communication message that is formatted by a server and returned to the original requester. The message contains the results of the server's work. See also request message.

reply translation header. A group of header fields that the Pathway translation server for the NonStop TUXEDO system inserts at the beginning of each reply message it relays from the application service to a SCREEN COBOL or Pathsend requester. This header indicates whether the request was processed successfully and whether the reply contains data returned by the application service. See also request translation header.

requester. A process or program that runs in the Guardian environment on a Tandem NonStop system and requests services from a server process. For example, a SCREEN COBOL program is a requester program that is interpreted by the terminal control process (TCP), which provides link access to Pathway server classes. Another type of requester program makes requests through Pathsend procedure calls; such a requester uses the LINKMON process for link access to server classes. A third type of requester communicates with server processes directly by calling the Guardian WRITEREAD procedure; this kind of requester does not use server classes. A requester is a specific type of client. See also client, server, and requester/server model.

requester/server model. A model for application design that divides the tasks of data input, data manipulation, and data output between two basic types of process: requesters and servers. A requester sends a request to a server. The server takes the requested action and then replies to the requester. The requester and server may reside on the same processor or on different processors. This model is used for interprocess communication in the Guardian environment. See also requester and server.

request message. The part of an interprocess communication message that is formatted by a requester and sent to a specific server. The message contains any data and instructions needed by the server to perform its processing. See also reply message.

request/response model. A model for requester/server communication in which a requester passes a single request to a server process and receives a single response. See also conversational model and request/response server.

request/response server. A server that provides request/response services. A service of type request/response is handled like a procedure and has the following properties: it is

executed until completion, it does not have any dialog with the requester, and it sends back a return value to the requester. A request/response server is analogous to a context-free Pathway server.

request translation header. A group of header fields that must be included at the beginning of each request message a SCREEN COBOL or Pathsend requester sends to the Pathway translation server for the NonStop TUXEDO system. This header specifies the NonStop TUXEDO application service for which the message is destined, the NonStop TUXEDO buffer types of the request and reply messages as seen by the service, and options that modify the invocation of the service. The translation server removes this header from the request message before sending it to the application service. See also reply translation header.

reserved word. A word that can be used only as a keyword.

resources. The components of a computer system that work together to process transactions. Terminals, workstations, CPUs, memory, I/O controllers, disk drives, processes, files, and applications are examples of resources.

response time. The amount of time it takes to receive a response from the system after initiating a request message (for example, by pressing a function key).

retryable operation. An operation that can be interrupted and repeated an indefinite number of times without affecting the consistency of the database; for example, all read operations are retryable.

RISC. See reduced instruction-set computing (RISC).

root directory. An OSS directory associated with a process that the system uses for pathname resolution when a pathname begins with a slash (/) character. See also OSS pathname.

RSC. See Remote Server Call (RSC).

scalability. The ability to increase the size and processing power of an online transaction processing system by adding processors and devices to a system, systems to a network, and so on, and to do so easily and transparently without bringing systems down. Scalability is also sometimes called expandability.

SCOBOLX. The object file for the SCREEN COBOL compiler program. This name is given in a TACL command to invoke the compiler. See also SCREEN COBOL.

screen. A group of data fields that represent formatted data to be displayed on a terminal. A screen is defined by a screen description entry in the Screen Section of a SCREEN COBOL program. There are two types of screen: base screens and overlay screens. See also base screen, overlay screen, and screen description entry.

SCREEN COBOL. A procedural language developed by Tandem and based on COBOL that is used to define and control screen displays on terminals and other input/output devices. SCREEN COBOL allows programmers to write requester programs that communicate with operator terminals and intelligent input/output devices, and that send data to server

processes that manage application databases. SCREEN COBOL programs are compiled into pseudocode form by the SCREEN COBOL compiler and then interpreted by the TCP. See also terminal control process (TCP).

SCREEN COBOL Utility Program (SCUP). A utility that provides control and manipulation of SCREEN COBOL object files.

screen description entry. A declaration of a base screen, and, optionally, an overlay screen, in the Screen Section of a SCREEN COBOL program. See also screen, base screen, and overlay screen.

screen-oriented requester. A SCREEN COBOL requester that sends data from working storage to the display screen of a terminal by way of screen templates defined in the Screen Section of the Data Division. Similarly, such a requester receives data from the terminal into working storage by way of Screen Section templates. It uses ACCEPT and DISPLAY statements in the Procedure Division to interact with the display terminals. Standard SCREEN COBOL requesters are screen-oriented. See also message-oriented requester.

screen overlay area. See overlay area.

screen program. A SCREEN COBOL requester program. See also SCREEN COBOL.

Screen Section. A section in the Data Division of a SCREEN COBOL source program that describes the types and locations of fields in screens that can be displayed on a terminal.

SCUP. See SCREEN COBOL Utility Program (SCUP).

SEND operation. In SCREEN COBOL, an operation in which a transaction request message is sent to a server process and a reply is received back from the server process. See also server-class send operation.

server. (1) A process or program that provides services to a client or a requester. Servers are designed to receive request messages from clients or requesters; perform the desired operations, such as database inquiries or updates, security verifications, numerical calculations, or data routing to other computer systems; and return reply messages to the clients or requesters. A server process is a running instance of a server program. (2) A combination of hardware and software designed to provide services in response to requests received from clients across a network. For example, Tandem's Himalaya servers provide transaction processing, database access, and other services. (In the NonStop TS/MP and Pathway/TS manual set, the word "server" is generally used only when definition 1 is meant; for definition 2, "system" is usually used instead of "server.") See also client, requester, client/server model, and requester/server model.

server class. A group of duplicate copies of a single server process, all of which execute the same object program. Server classes are configured through the PATHMON process.

server-class send operation. The sending of a message to a Pathway server class by making a call to a Pathsend procedure. The SERVERCLASS_SEND_,

SERVERCLASS_DIALOG_BEGIN_, and SERVERCLASS_DIALOG_SEND_ procedures perform server-class send operations. See also SEND operation.

SERVER object. A definition of a server class within the configuration of a PATHMON process.

service. A function performed by a server process or program on behalf of a requester or client. A server can perform one or several services. The concept of a service is built into the design of the BEA TUXEDO system and the NonStop TUXEDO system; for these products, a service is a module of application code that carries out a service request.

simple token. In the Subsystem Programmatic Interface (SPI), a token consisting of a token code and a value that is either a single elementary field, such as an integer or a character string, or a fixed (nonextensible) structure. See also extensible structured token and token (definition 2).

single-threaded. A programming model that provides a single thread of control within a program. For example, a single-threaded server handles only one request at a time and must complete that request before accepting another. See also thread and multithreaded.

special register. A data item defined by the SCREEN COBOL compiler, rather than explicitly in the program. Each special register has a particular purpose and should be used only as defined. The SCREEN COBOL language defines a different set of special registers from those defined by the standard COBOL language.

SPI. See Subsystem Programmatic Interface (SPI).

static server. A server process that the PATHMON process creates when a START SERVER command is issued. The PATHMON process starts the number of static servers defined by the NUMSTATIC attribute for the server class. See also dynamic server.

Structured Query Language (SQL). A relational database language used to define, manipulate, and control databases.

subsystem. In the context of the Subsystem Programmatic Interface (SPI) and the Event Management Service (EMS), a process or collection of processes that gives users access to a set of related resources or services. A subsystem typically controls a cohesive set of objects. The Pathway subsystem includes PATHMON processes, TCPs, and all other components of the PATHMON environment.

subsystem ID. In management applications, a data structure that uniquely identifies a subsystem. The subsystem ID is specified in SPI commands to identify the target subsystem, and in EMS event messages to identify the source of the event message. The Pathway subsystem ID applies to all components of the PATHMON environment, including PATHMON processes and TCPs.

Subsystem Programmatic Interface (SPI). A set of Guardian procedures, message formats, and definition files that allows management applications to communicate directly with

subsystem processes, such as the PATHMON process, for configuration and control of objects and for event management.

subtype 30 process. A nonprivileged process that simulates terminals and communications devices.

subvolume. A related set of files, as defined by the user, within the Guardian environment. The name of a subvolume is the third of the four parts of a file name.

swap file. A temporary file created by a process for temporary storage: for example, by the Kernel Managed Swap Facility (KMSF) on behalf of a TCP for temporary storage of terminal context.

syncdepth. A parameter to Guardian procedure calls that sets the maximum number of operations or messages that a process is allowed to queue before action must be taken or a reply must be performed.

sync ID. A value used in the Guardian environment to determine whether an I/O operation has finished. In active backup programming, a file's sync ID is used to prevent the backup process from repeating I/O operations that have already been completed by the primary process.

system name. (1) The first of the four parts of a Guardian file name; also called a node name. (2) A name that identifies the Tandem system on which a PATHMON process is running. In SCREEN COBOL programs, this name is given in SEND statements. (3) A SCREEN COBOL word that identifies part of the Tandem operating environment; a name can be associated with function keys or terminal display attributes.

System/T. The transaction monitor for a NonStop TUXEDO system. See also NonStop TUXEDO application.

TACL. See Tandem Advanced Command Language (TACL).

TAL. See Transaction Application Language (TAL).

Tandem Advanced Command Language (TACL). The user interface to the Tandem NonStop Kernel in the Guardian environment. The TACL product is both a command interpreter and a command language.

Tandem Alliance. Tandem's marketing and technical support program for third-party vendors, which encourages the development of application software for the Pathway environment. The Alliance attracts software developers, value-added resellers, and other vendors who can provide industry-specific and general business applications for Tandem customers.

Tandem NonStop Kernel. The operating system for Tandem NonStop systems. The operating system does not include any application program interfaces.

Tandem NonStop Series (TNS). Tandem computers that support the Tandem NonStop Kernel and that are based on complex instruction-set computing (CISC) technology.

TNS processors implement the TNS instruction set. See also complex instruction-set computing (CISC) and Tandem NonStop Series/RISC (TNS/R).

Tandem NonStop Series/RISC (TNS/R). Tandem computers that support the Tandem NonStop Kernel 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. See also reduced instruction-set computing (RISC) and Tandem NonStop Series (TNS).

task. The sequence of SCREEN COBOL program units that are executed as a result of a PATHCOM START TERM or RUN PROGRAM command or an SPI START TERM or START PROG command.

TCLPROG file. A SCREEN COBOL object library file.

TCP. See terminal control process (TCP).

TDA. See terminal data area (TDA).

TEDIT. A Tandem text editor used to create or modify a source text file. Also called PS Text Edit.

tell message. An informational message sent by PATHCOM or a management application to one or more terminals controlled by a SCREEN COBOL program, to be displayed for the terminal operators.

TELL object. A temporary object used in PATHCOM and SPI commands to define a tell message.

temporary TERM object. A TERM object created by the PATHMON process when a PATHCOM RUN PROGRAM command or an SPI START PROG command is issued. Temporary TERM objects are deleted by the PATHMON process when application processing is completed or when a STOP TERM or ABORT TERM command is issued. Names of temporary TERM objects begin with a number. See also configured TERM object and TERM object.

terminal. An I/O device capable of sending and receiving information over communications lines.

terminal context. Data maintained by a TCP for each active terminal under its control.

terminal control process (TCP). A process used for terminal management and transaction control, provided by Tandem as part of the Pathway/TS product. A TCP is a multithreaded process that interprets compiled SCREEN COBOL requester programs (screen programs) in the user’s application, executing the appropriate program instructions for each I/O device or process the TCP is configured to handle. The TCP coordinates communication between screen programs and their I/O devices or processes and, with the help of the PATHMON process, establishes links between screen programs and server processes. See also requester and SCREEN COBOL.

terminal data area (TDA). In SCREEN COBOL, the area that the TCP allocates for terminal context data. The MAXTERMDATA parameter of the PATHCOM SET TCP command defines the upper limit for this data area.

TERM object. A definition of a task that uses a SCREEN COBOL program to control an input/output device such as a terminal or workstation, or an input/output process such as a front-end process. A TERM object can be either explicitly configured with an ADD command or created by the PATHMON process through a PATHCOM RUN PROGRAM or SPI START PROG command. TERM objects created by the latter method are called temporary TERM objects. See also configured TERM object and temporary TERM object.

thaw condition. A condition in which prohibition of communication between a terminal and a server class is lifted. See also freeze condition.

thread. A task that is separately dispatched and that represents a sequential flow of control within a process (for example, a TCP).

throughput. The number of transactions a system can process in a given period, such as one second.

TMF. See Transaction Management Facility (TMF) subsystem.

TMF level recovery. Recovery of the database to a consistent state through the use of the TMF subsystem. When a failure occurs, the TMF subsystem allows the application to back out the entire transaction, returning the contents of the database to the values it held when the transaction was started. The application can then retry the transaction.

TNS. See Tandem NonStop Series (TNS).

TNS/R. See Tandem NonStop Series/RISC (TNS/R).

token. (1) An attribute control element in the CONTROLLED clause of a SCREEN COBOL program, which allows run-time control of display attributes. This token consists of an attribute identifier and an attribute value. (2) In the Subsystem Programmatic Interface (SPI), a distinguishable unit in a message. An SPI token consists of an identifier (token code or token map) and a token value. Programs place tokens in an SPI buffer by calling the SSPUT procedure and retrieve them from the buffer by using the SSGET procedure.

transaction. An operation or a series of operations that retrieves and updates information to reflect an exchange of goods or services. In the process of retrieving and updating information, a transaction transforms a database from one consistent state to another. The TMF subsystem treats a transaction as a single unit; either all of the changes made by a transaction are made permanent (the transaction is committed) or none of the changes are made permanent (the transaction is aborted).

Transaction Application Language (TAL). A systems programming language with many features specific to stack-oriented Tandem NonStop systems. See also Portable Transaction Application Language (pTAL).

transaction backout. A TMF subsystem activity in which the effects of a partially completed transaction are canceled.

Transaction Delivery Process (TDP). A multithreaded gateway process, part of the Remote Server Call (RSC) product, that runs on a Tandem NonStop system. The TDP can be replicated and can manage many connections and workstations at one time, as well as multiple sessions from each workstation.

transaction identifier. A unique name that the TMF subsystem assigns to a transaction.

transaction log. See audit trail.

transaction management. The ability to coordinate transaction control functions, such as beginning and ending transactions, committing or aborting transactions, and recovering transactions.

Transaction Management Facility (TMF) subsystem. The major component of the NonStop TM/MP product, which protects databases in online transaction processing environments. To furnish this service, the TMF subsystem manages database transactions, keeps track of database activity through audit trails, and provides database recovery methods. See also NonStop Transaction Manager/MP (NonStop TM/MP) and transaction.

transaction mode. The operating mode of a requester program between the execution of a BEGINTRANSACTION procedure call or statement and the execution of the associated ENDTRANSACTION or ABORTTRANSACTION call or statement.

transaction processing. See online transaction processing (OLTP).

transaction workload. The number of transactions to be processed by an online transaction processing application.

TRANSFER delivery system. An information delivery system that enables organizations to move and manage information efficiently within a single Tandem system or across a network of systems. The TRANSFER delivery system supports communications between users, I/O devices, and processes in the Guardian environment.

TSCODE. The object code for the part of a GDSX process that is supplied by Tandem. TSCODE includes generic routines and services that support the development of a multithreaded, fault-tolerant front-end process. See also USCODE and Extended General Device Support (GDSX).

tuning. The process by which a system manager allocates and balances resources for optimum system performance.

UMP. See unsolicited-message processing (UMP).

unsolicited message. A message that is sent to a SCREEN COBOL program and includes application-dependent information to be processed by the program. Although the program does not do anything to initiate the message, the message must conform to the

format defined by the program. The message is sent first to the TCP. It contains a header with information that is used by the TCP and a body with information that the TCP delivers to the SCREEN COBOL program.

unsolicited-message processing (UMP). The feature that allows terminals running SCREEN COBOL requesters to accept and reply to unsolicited messages sent to them by Guardian processes outside of the PATHMON environment.

USCODE. The object code for the part of a GDSX process that is developed by the user to provide device or access-method specifics such as control operations or data-stream translation. USCODE is bound with TSCODE to produce an operational GDSX process. See also TSCODE and Extended General Device Support (GDSX).

user conversion procedure. A procedure that lets users make their own validation checks or conversions of data passed between a SCREEN COBOL program and a terminal screen or intelligent device.

ViewPoint application. An extensible interactive application for managing operations in the Guardian environment. It provides tools for interacting with multiple Tandem subsystems, including PATHCOM, allowing a system manager to easily control an integrated Tandem system from one location.

volume. A disk drive, or a pair of disk drives that forms a mirrored disk, in the Guardian environment. The name of a volume is the second of the four parts of a file name.

$RECEIVE. A special Guardian file name through which a process receives and optionally replies to messages from other processes by using Guardian procedure calls. This file is analogous to a request queue defined for a NonStop TUXEDO server.

Index

A

Aborted dialog (error 929) 6-12
Aborted dialogs, detecting 4-16/4-17
Aborted send operation (error 918) 6-9
Aborted transaction (error 934) 6-14
Aborting transactions 2-8, 2-23, B-15
ABORTTRANSACTION procedure 4-6
Access Control Server (ACS) 3-7
Application development tools 1-3
Application generator, Pathmaker 1-14
Application Transaction Monitor Interface (ATMI) 3-11
Applications, designing
- batch processing 2-27
- database 2-9/2-10
- example, OLTP 2-1/2-8
- requester programs 2-11/2-19
- server programs 2-19/2-27
- transactions 2-1/2-8

Applications, Pathway
- client/server capabilities 1-12
- data integrity 1-4
- development of 1-3/1-4, 1-14/1-15
- distributed processing 1-7
- expansion fundamentals 1-7
- fault tolerance 1-5/1-6
- introduction 1-1
- managing 1-4
- overview 1-7
- performance of 1-6, 1-8
- requester programs 1-9/1-10
- security fundamentals 1-6
- server classes 1-8
- server languages 1-8
- server processes 1-8
- support for NonStop TUXEDO environment 1-13

Applications, Pathway (continued)
- transaction processing scenario 1-15/1-17

Applications, programming
- examples
  - nested server program B-53/B-67
  - Pathsend requester program B-1/B-52
  - skeleton Pathsend program 2-14/2-15
  - skeleton server program 2-25/2-27
- for TMF subsystem 4-5/4-10
- invoking Pathsend procedures
  - from C and C++ programs 5-2
  - from COBOL85 programs 5-3
  - from Pascal programs 5-4
  - from TAL and pTAL programs 5-5
- overview 1-1
- Pathsend environment 1-10/1-11
- Pathsend requesters
  - See Requesters, writing Pathsend
- Pathsend usage considerations 5-23/5-28
- Pathway servers
  - See Server programs, writing
- writing Pathsend requesters 3-1/3-11
- writing Pathway servers 4-1/4-17

Application-Transaction Monitor Interface (ATMI) 1-13
ASSIGNs 3-7, B-5, B-12, B-18, B-55
ATM devices 2-16
ATMI functions, NonStop TUXEDO 1-13, 3-11
Audit trails, TMF 1-4, 2-8, 4-5
Audited and nonaudited servers 2-24
Audited files 4-7
Automatic retry, Pathsend requesters 1-9, 3-6
Autonomy, node 1-8

AWAITIOX procedure 5-7, 5-13, 5-17, 5-25, 5-28

B

Back-end process 2-16, 2-24
BASIC, Extended, for Pathway servers 1-8
Batch processing 1-7, 2-13, 2-27
Bounds error (error 912) 6-6
Buffer length invalid (error 911) 6-6
Buffer limits A-1

C

C and C++ languages
- for Pathsend requesters 1-10
- for Pathway servers 1-8
- invoking Pathsend procedures 5-2

Calls
- See Procedure calls, Pathsend, individual procedures

CANCEL procedure 5-25/5-26
Canceling server-class send calls 5-25/5-26
CANCELREQ procedure 5-25/5-26
Changed transid (error 930) 6-13
Checkpointing
- and Pathsend 3-6
- explanation of 1-5
- Pathsend limitations 2-14
- server considerations 4-3

Classes of data in database 2-9
Clients, NonStop TUXEDO
- interoperating with 4-17
- Pathsend requesters acting as 3-11

Client/server computing 2-15
- capabilities 1-12
- development tools 1-15

COBOL85
- example Pathsend requester program B-53/B-67
- example server program B-53/B-67
- for Pathsend requesters 1-10
- for Pathway servers 1-8
- invoking Pathsend procedures 5-3

Code, standardizing and testing 1-3
Colon, in Pathsend procedure parameters 5-2
Common Run-Time Environment (CRE) 4-3
Compilers provided 1-14
Concurrent processing 2-7/2-8, 4-5
Condition code
- considerations 5-23
- register 5-2

Context-free
- servers, using with context-sensitive requesters 3-8, 4-4

Context-sensitive programming, requesters
- canceling server-class sends 3-10
- failure recovery 3-9/3-10
- overview 3-8
- resource utilization 3-8
- using with context-free servers 3-8, 4-4

Context-sensitive programming, servers
- controlling dialogs 4-14
- correlating messages with dialogs 4-16
- detecting aborted dialogs 4-16/4-17
- detecting new dialogs 4-14
- functions performed 4-13
- handling dialog messages 4-14/4-15
- managing dialogs 4-16
- overview 4-13

Control block examples B-8, B-13, B-14, B-23, B-25
Conversational mode 2-12
Conversational servers 4-1
CPU halts and LINKMON processes 4-3

CRE (Common Run-Time Environment) 4-3
Creation failure, servers (error 916) 6-8
Creator default for PATHMON names 3-7
CRE_Receive_Read_ procedure 4-14
Crossref product 1-14

D

Data
- analyzing flow of 2-2/2-3
- classes of 2-9
- integrity 1-4

Database
- concurrency 2-7/2-8
- consistency 1-4, 2-7/2-8
- consistency and concurrency 4-5
- fields in 2-9
- files in, normalizing 2-9
- integrity 1-8
- logical design 2-9
- management systems 2-10
- physical design 2-10
- records in 2-9
- relational 2-10, 4-5

DBMS (database management system)
- See Database

Deadlocks
- for transactions 4-10
- with nested servers 2-22

Debugging
- Pathway servers 4-11/4-12
- tools for 1-14

Declarations, global B-9
Declaratives, example B-58

Design
- application example 2-1/2-8
- batch processing applications 2-27
- database 2-9/2-10
- requester programs 2-11/2-19
- server programs 2-19/2-27
- transactions 2-1/2-8

Detecting aborted dialogs 4-16/4-17

Development
- considerations 1-3/1-4
- tools 1-14/1-15

Dialog abort system message (message -121) 4-16

Dialogs
- aborted (error 929) 6-12
- continuing, aborting, terminating 4-16
- controlling 4-14
- correlating messages with 4-16
- detecting aborted 4-16/4-17
- detecting new 4-13, 4-14
- ended (error 931) 6-13
- handling messages 4-14/4-15
- invalid (error 926) 6-11
- models, types of 3-9
- model, specifying 5-10, 5-15
- outstanding (error 933) 6-13
- too many (error 927) 6-12
- using TMF subsystem 4-15

Distributed processing 1-6, 1-7
Distributed transaction processing (DTP) 1-7
Dynamic links 3-4

E

Early replies 2-23
Enscribe product 2-10, 4-5
Entry-sequenced files 2-10
Errors
  See also Errors, file-system; Failure recovery; individual errors
  dialog abort system message (message -121) 4-16
  FESCErr (error 233) 5-6, 5-7, 5-12, 5-13, 5-17
  handling Pathsend 6-1/6-14
  Pathsend server TIMEOUT 5-27
  returned by servers in replies 4-14
  server process 3-4
Errors, file-system
  associated with Pathsend errors 6-1/6-14
  FEBoundsErr (error 22) 5-21
  FEContinue (error 70) 4-14, 5-7, 5-13
  FEEOF (error 1) 4-14
  FEInUse (error 12) 6-3
  FEInvalOp (error 2) 5-21
  FEMissParam (error 29) 5-21
  FENoBufSpace (error 31) 6-9
  FENoDiscSpace (error 43) 6-14
  FENoSuchDev (error 14) 6-3, 6-4
  FEOK (error 0) 4-14
  FEPathDown (error 201) 6-3, 6-5
  FESecViol (error 48) 6-3, 6-4
  FETimeout (error 40) 6-3
  FETimeout (error 440) 6-4
  Requesting process has no… (error 75) 4-6
  timeout error (error 40) 4-11/4-12
Event Management Service (EMS) 1-4
Examples
  nested server program B-53/B-67
  Pathsend requester program B-1/B-52
Extended BASIC, for Pathway servers 1-8
Extended General Device Support (GDSX) processes 1-10, 2-16/2-18, 2-24

F

Failure recovery
  context-sensitive requesters 3-9/3-10
  fault-tolerant programming
    requesters 3-5/3-6
    servers 4-10
  LINKMON limit errors 3-4
  LINKMON process 4-3
  overview 3-3/3-4
  Pathsend and TMF subsystem 3-5
  Pathsend requester 4-2
  server process 3-4
  servers and TMF subsystem 4-6/4-10
Fault tolerance
  and servers using the TMF subsystem 2-23
  overview 1-5, 1-6
  Pathsend requester programming 3-5/3-6
  Pathway server programming 4-10
FEBoundsErr (error 22) 5-21
FEContinue (error 70) 3-10, 4-14, 5-7, 5-13
FEEOF (error 1) 3-9, 4-14
FEInUse (error 12) 6-3
FEInvalOp (error 2) 5-21
FEMissParam (error 29) 5-21
FENoBufSpace (error 31) 6-9
FENoDiscSpace (error 43) 6-14
FENoSuchDev (error 14) 6-3, 6-4
FEOK (error 0) 4-14
FEPathDown (error 201) 6-3, 6-5
FEScChangedTransid (error 930) 6-13
FEScDialogAborted (error 929) 3-4, 3-9, 6-12
FEScDialogEnded (error 931) 6-13
FEScDialogInvalid (error 926) 6-11
FEScDialogOutstanding (error 933) 6-13

FESCErr (error 233) 5-6, 5-7, 5-12, 5-13, 5-17
FEScError (error 233) 3-9
FEScInvalidBufferLength (error 911) 6-6
FEScInvalidFlagsValue (error 909) 6-6
FEScInvalidPathmonName (error 901) 6-2
FEScInvalidSegmentId (error 907) 6-5
FEScInvalidServerClassName (error 900) 6-2
FEScInvalidTimeoutValue (error 919) 6-9
FEScLinkmonConnect (error 947) 6-14
FEScMissingParameter (error 910) 6-6
FEScNoSegmentInUse (error 908) 6-5
FEScNoSendEverCalled (error 906) 6-5
FEScNoServerLinkAvailable (error 905) 4-3, 6-5
FEScOutstandingSend (error 928) 6-12
FEScParameterBoundsError (error 912) 6-6
FEScPathmonConnect (error 902) 4-11/4-12, 6-3
FEScPathmonMessage (error 903) 6-4
FEScPathmonShutDown (error 915) 4-11/4-12, 6-7
FEScPFSUseError (error 920) 6-9
FEScSendOperationAborted (error 918) 6-9
FeScSendOperationAborted (error 918) 5-27
FEScServerClassFrozen (error 913) 6-7
FEScServerClassTmfViolation (error 917) 3-5, 6-8
FEScServerCreationFailure (error 916) 6-8
FEScServerLinkConnect (error 904) 3-4, 6-4
FEScTooManyDialogs (error 927) 6-12
FEScTooManyPathmons (error 921) 6-9
FEScTooManyRequesters (error 925) 6-11
FEScTooManySendRequests (error 924) 6-11
FEScTooManyServerClasses (error 922) 6-10
FEScTooManyServerLinks (error 923) 4-3, 6-10
FEScTransactionAborted (error 934) 6-14
FEScUnknownServerClass (error 914) 6-7
FESecViol (error 48) 6-3, 6-4
FETimeout (error 40) 6-3, 6-4
Fields, database 2-9
FILEINFO 5-26
Files
  database 2-9/2-10
  entry-sequenced 2-10
  I/O, in SCREEN COBOL requesters 2-12
  key-sequenced 2-10
  relative 2-10
  unstructured 2-10
File-number parameter 5-26
File-system errors
  See Errors, file-system
FILE_GETRECEIVEINFO_ procedure 4-14, 4-16
flag parameter 3-9
Flags, invalid value (error 909) 6-6
FORTRAN, for Pathway servers 1-8
Front-end process 2-16/2-18
Frozen server class (error 913) 6-7
Fundamentals of Tandem NonStop systems 1-3/1-7

G

GDSX (Extended General Device Support) processes 1-10, 2-16/2-18, 2-24
Guardian operating environment
  distributed processing in 1-7
  processes in 1-5
  security features of 1-6
  servers in 1-8, 2-19

H

Halts, CPU 4-3

I

IDS
    See Intelligent device support (IDS)
Industrial robots 2-16
Input file structure example B-6
Inspect product 1-14
Intelligent device support (IDS)
    description 2-12
    GDSX programming for 2-17
    overview 1-7
    RSC requesters 1-12, 2-15
Intelligent mode 2-12
Interoperation
    of Pathsend requesters with NonStop TUXEDO servers 1-13
    of Pathway servers with NonStop TUXEDO requesters (clients) 1-13
    with NonStop TUXEDO requesters (clients) 4-17
    with NonStop TUXEDO servers 3-11
Interprocess communication, Pathsend 1-11, 3-2
Invalid buffer length (error 911) 6-6
Invalid dialog (error 926) 6-11
Invalid flagsValue (error 909) 6-6
Invalid PATHMON name (error 901) 6-2
Invalid server class name (error 900) 6-2
Invalid timeout value (error 919) 6-9
I/O
    cancel outstanding, example B-19
    complete outstanding, example B-20
    starting example B-13

K

Key field, database 2-9
Key-sequenced files 2-10

L

Languages supported
    Pathsend requesters 1-10
    Pathway servers 1-8
Limits, Pathsend environment A-1
link denied (error 4) 4-3
Link management, with LINKMON process 1-11, 2-13
LINKMON process
    connect error (error 947) 6-14
    description 1-11
    failures 4-3
    GDSX, relationship to 2-17/2-18
    limits errors 3-4
    RSC, relationship to 1-12, 2-15
    servers, relationship to 4-2
    TMF transaction identifiers 5-24
Links
    allocating space for 4-3
    server connect error (error 904) 6-4
    static and dynamic 3-4
    too many server links (error 923) 6-10
Locking of records 4-8, 4-10

M

Manageability
    of Pathway applications 1-4
    provided by server classes 1-8
Management interfaces 1-8, 2-19
    See also PATHCOM interface, SPI (Subsystem Programmatic Interface)
Message buffer
    SERVERCLASS_DIALOG_BEGIN_ procedure 5-8
    SERVERCLASS_DIALOG_SEND_ procedure 5-14
    SERVERCLASS_SEND_ procedure 5-18

Messages
- dialog abort system message (message -121) 4-16
- error, checking for 2-14
- example print to terminal and abend B-15
- for checkpointing 1-5
- handling in a dialog 4-14/4-15
- processing example B-60
- receiving by servers 4-2
- reply size, specifying maximum 5-9, 5-14, 5-19
- unrecognizable (error 903) 6-4

Missing parameter (error 910) 6-6
Modes, terminal 2-12
Modularity 1-8
Multiprocessing 1-6
Multithreading
- GDSX feature 2-16
- in server design 2-20
- Pathsend requesters 1-9

N

Native System /T clients, NonStop TUXEDO, interoperating with 4-17
Nested servers 4-4
- deadlocks 2-22
- designing 2-22/2-23
- example program B-53/B-67
- overview 2-13

Network security, Pathsend requesters 3-6
No send ever called (error 906) 6-5
No server link available (error 905) 6-5
Node autonomy 1-8
Nonaudited files 4-7
Nonretryable requests, Pathsend 3-6
NonStop Kernel Open System Services (OSS) server processes
- See Open System Services (OSS) server processes

NONSTOP parameter 4-10
NonStop SQL/MP (Structured Query Language/MP) product 2-10, 4-5
NonStop systems 1-5/1-7
NonStop Transaction Manager/MP (NonStop TM/MP)
- advantages 1-3
- application restrictions 4-7
- audited files 4-7
- context-free Pathsend requesters 3-5
- context-sensitive requesters 3-9/3-10
- dialogs, using with 4-15
- fault-tolerant servers 4-10
- grouping transaction operations 4-8/4-9
- limits A-1
- Pathmaker software 1-15
- Pathsend procedure calls 3-5
- record locking 4-8
- requester example B-1/B-52
- retryable operations 4-7
- server application structure 4-5/4-6
- server class violation (error 917) 6-8
- server process pairs 2-23
- TMF OFF server parameter 3-5
- transaction deadlocks 4-10
- usage considerations 5-24
- writing servers to use 4-5/4-10

NonStop Transaction Services/MP (NonStop TS/MP) product xi, 1-1, 1-3
NonStop TUXEDO
- applications, interoperation with 1-13, 3-11, 4-17
- requesters (clients), writing Pathway servers for 1-13, 4-17
- servers, writing Pathway requesters for 3-11

Nowait send operations
  errors for 6-1
  limit on number of A-1
  procedure calls for 5-7, 5-13, 5-17
  specifying 5-9, 5-15, 5-20
  usage considerations 5-23/5-24, 5-28
No-early-reply rule 2-23

O

OLTP
  application design example 2-1/2-8
  development considerations 1-3/1-4
  expanding systems for 1-7
  importance of fault tolerance for 1-5
  manageability 1-4
  Pathway environment 1-3/1-7
  support for NonStop TUXEDO environment 1-13
  transaction processing scenario 1-15/1-17
Online transaction processing (OLTP)
  See OLTP
Open System Services (OSS) processes 3-11
  Pathway servers 2-19
Open System Services (OSS) server processes 1-8, 2-19, 3-11
Operation number, server-class 5-24/5-26
Output record example B-6
Outstanding send (error 928) 6-12
OWNER attribute, server 3-7

P

Parameter bounds error (error 912) 6-6
Parameter missing (error 910) 6-6
Parameter pairs, in Pathsend procedure syntax 5-2
PARAMs example B-6, B-13
Pascal
  for Pathsend requesters 1-10
  for Pathway servers 1-8
  invoking Pathsend procedures 5-4
PATHCOM interface
  description 1-4
  use in managing servers 1-8, 2-19
Pathmaker product 1-14
PATHMON process
  avoiding coded names 3-7
  connect error (error 902) 6-3
  description 1-4
  fault-tolerance role 1-6
  invalid name (error 901) 6-2
  name, specifying 5-18
  relationship to LINKMON processes 1-11
  shutdown (error 915) 6-7
  too many (error 921) 6-9
  unrecognizable message (error 903) 6-4
Pathsend application program interface (API)
  See also Requesters, writing Pathsend; individual procedures
  error handling 6-1/6-14
  example nested server program B-53/B-67
  example requester program B-1/B-52
  failures, LINKMON 4-3
  failures, requester 4-2
  interprocess communication 1-11, 3-2
  limits, programming environment A-1
  LINKMON processes, relationship to 1-11
  nonretryable requests 3-6
  processes using 1-10
  programming languages supported 1-10
  requesters, overview 1-9
  timeout considerations 5-27/5-28

Pathsend application program interface (API) (continued)
- transaction processing scenario 1-15/1-17

Pathsend procedure calls
- See also individual procedures
- errors returned 6-1/6-14
- invoking from C and C++ programs 5-2
- invoking from COBOL85 programs 5-3
- invoking from Pascal programs 5-4
- invoking from TAL and pTAL programs 5-5
- overview 5-1
- return errors 6-1
- usage considerations 5-23/5-28
- use by NonStop TUXEDO requesters 4-17

Pathsend requesters
- checkpointing limitations 2-14
- description 1-9
- design considerations 2-13/2-14
- program structure 2-14/2-15

Pathway applications
- See Applications, Pathway; Applications, programming; Applications, designing

Pathway environment
- See also Applications, programming; Applications, designing
- advantages of 1-3/1-7
- products for 1-1, 2-11

Pathway servers
- See also Server classes; Server processes; Server programs; Applications, programming; Applications, designing
- description 1-7

Pathway to TUXEDO translation server 1-13, 3-11

Pathway/TS product xi, 1-1, 1-3

Performance
- provided by Pathsend requesters 1-9
- provided by Pathway applications 1-6
- provided by server processes 1-8

Personal computer support 2-15

PFS use error (error 920) 6-9

POET (Pathway Open Environment Toolkit) product 1-3, 1-12, 2-16

Portable Transaction Application Language
- See pTAL

POS (point-of-sale) devices 2-16

Presentation services 1-7

Procedure calls, Pathsend
- See also individual procedures
- errors returned 6-1/6-14
- invoking from C and C++ programs 5-2
- invoking from COBOL85 programs 5-3
- invoking from Pascal programs 5-4
- invoking from TAL and pTAL programs 5-5
- overview 3-1, 5-1
- return errors 6-1
- summary 5-1
- usage considerations 5-23/5-28

Process pairs 1-5, 2-23

Processes
- description 1-5
- distribution of 1-6
- primary and backup 1-5
- replication of 1-6
- starting up example B-4

Programming
- See Applications, programming; Applications, designing

Programming languages
- Pathsend requesters 1-10
- Pathway servers 1-8

PROGRAM-STATUS special register 4-10

pTAL
- for Pathsend requesters 1-10
- for Pathway servers 1-8
- invoking Pathsend procedures 5-5

PWY2TUX translation server 1-13, 3-11

Q

Queuing of incomplete transactions 2-23

R

RDBMS (relational database management system)
- See Database management systems

RDF
- See Remote Duplicate Database Facility (RDF)

Read operations, repeatable 4-8

RECEIVE messages
- allocating space 4-3
- determining new dialogs 4-13

Record locking 4-8, 4-10

Records, database 2-9

Recovery
- context-sensitive requesters 3-9/3-10
- fault-tolerant programming
  - requesters 3-5/3-6
  - servers 4-10
- LINKMON limit errors 3-4
- LINKMON process 4-3
- Pathsend and TMF subsystem 3-5
- Pathsend requester failures 4-2
- requester overview 3-3/3-4
- server process failures 3-4
- servers and TMF subsystem 4-5/4-10

Relational database management 2-10, 4-5

Relative files 2-10

Remote Duplicate Database Facility (RDF) 2-11

Remote Server Call
- See RSC (Remote Server Call)

Repeatable requests, Pathsend 3-6

Reply format, server program 4-1

Requesters
- See also Applications, programming; Applications, designing; Requesters, writing Pathsend
- clients using POET 2-16
- clients using RSC 2-15
- debugging 1-14
- description 1-7
- designing 2-11/2-19
- dividing functions with servers 2-19
- Pathmaker, using to develop 1-14
- Pathsend
  - description 1-9
  - design considerations 2-13/2-14
  - limits A-1
  - nested servers 2-22/2-23
  - program structure 2-14/2-15
- SCREEN COBOL
  - description 1-10
  - IDS 2-12
  - standard 2-12
- too many(error 925) 6-11
- transaction processing scenario 1-15/1-17
- types of 1-9, 2-11
- using GDSX 2-16/2-18

Requesters, writing Pathsend
- automatic retry 3-6
- avoiding coded PATHMON names 3-7
- context-sensitive programming 3-8/3-10
- example program B-1/B-52
- failure recovery overview 3-3/3-4
- fault-tolerant programming 3-5/3-6
- interprocess communication 3-2
- LINKMON limit errors 3-4

Requesters, writing Pathsend (continued)
- NonStop TUXEDO, interoperating with 3-11
- overview 3-3
- procedure calls,list of 3-1
- retryable requests 3-6
- security issues 3-6/3-7
- server process failures 3-4
- sharing servers 4-1
- TMF subsystem 3-5
- using ASSIGNs 3-7

Requesting process has no… (error 75) 4-6
Request/response servers 4-1
Resource utilization 3-8
Response time 1-6
Retry mechanism, Pathsend calls 3-5
- See also Fault tolerance

Retryable requests, Pathsend 3-6
Retryable server operations, TMF 4-7
Return errors 6-1
Robots, industrial 2-16
RSC (Remote Server Call)
- role in Pathway environment 1-12, 2-15
- security issues 3-7
- server programming considerations 4-4

S

SCF (Subsystem Control Facility), use in managing GDSX processes 2-18
SCREEN COBOL
- context sensitivity not supported 4-13
- debugging 1-14
- devices supported 2-12
- Pathmaker application generator 1-14
- requesters
  - description 1-10
  - designing 2-12/2-13
  - GDSX alternative 2-18
- server programming considerations 4-3
- unsolicited message processing (UMP) 2-12

Screen programs 2-12
scsend-op-num parameter 5-10, 5-15, 5-20, 5-24/5-26
Security
- for Pathsend requesters 2-14
- Pathsend programming issues 3-6/3-7
- system 1-6

SECURITY attribute, server 3-7
Send operation aborted (error 918) 6-9
Send operation outstanding (error 928) 6-12
Send requests, too many (error 924) 6-11
Server class frozen (error 913) 6-7
Server class unknown (error 914) 6-7
Server classes
- accessing
  - See Requesters
- description 1-8
- fault tolerance role 1-5
- limits A-1
- names, specifying 5-8, 5-18
- security for Pathsend requesters 3-7
- send operation number 5-10, 5-15, 5-20, 5-24/5-26
- timeouts, specifying 5-27
- TMF violation (error 917) 6-8

Server processes
- benefits 1-8
- description 1-7
- transaction integrity 1-5
- transaction processing scenario 1-15/1-17

Server programs
- aborting transactions 2-23
- audited and nonaudited 2-24
- debugging 1-14
- description 1-7
- designing 2-19/2-27

Server programs (continued)
- dividing functions with requesters 2-19
- example nested server B-53/B-67
- GDSX back-end process, using 2-24
- languages for 1-8
- nested 2-13
- nested servers 2-22/2-23
- no-early-reply rule 2-23
- packaging individual functions 2-21/2-22
- Pathmaker, using to develop 1-14
- single-threaded and multithreaded 2-20
- structure of 2-25/2-27
- TMF subsystem and fault tolerance 2-23
- utilization of 2-20

Server reply code
- for RSC clients 4-4
- for SCREEN COBOL requesters 4-3

SERVERCLASS_DIALOG_ABORT_ procedure
- overview 3-8
- syntax and usage 5-6

SERVERCLASS_DIALOG_BEGIN_ procedure
- canceling 5-25/5-26
- flag parameter 3-9
- overview 3-8
- syntax and usage 5-7/5-11
- TMF considerations 5-24

SERVERCLASS_DIALOG_END_ procedure
- overview 3-8
- syntax and usage 5-12

SERVERCLASS_DIALOG_SEND_ procedure
- canceling 5-25/5-26
- syntax and usage 5-13/5-16
- TMF considerations 5-24

SERVERCLASS_SEND_ procedure
- canceling 5-25/5-26
- example B-31, B-56, B-65
- nowait considerations 5-23/5-24, 5-28
- overview 3-3
- syntax and usage 5-17/5-20
- TMF considerations 5-24
- waited considerations 5-23, 5-27

SERVERCLASS_SEND_INFO_ procedure
- example B-17, B-56, B-66
- overview 3-3
- syntax and usage 5-21/5-22

Servers
- See Server classes; Server processes; Server programs; Applications, programming; Applications, designing
- canceling sends to server classes 3-10
- context-sensitive
  - errors returned 4-14
  - functions of 4-13
- context-sensitive requesters, using 3-8, 4-4
- conversational (context-sensitive) 4-1
- creation failure (error 916) 6-8
- debugging 4-11/4-12
- failures and Pathsend 3-4, 4-2
- interoperation with TUXEDO requesters 4-17
- link error (error 904) 6-4
- nested 4-4
- no link available (error 905) 6-5
- NonStop TUXEDO 3-11
- PWY2TUX translation server 3-11
- request/response (context-free) 4-1
- timeouts, specifying 5-27
- too many links (error 923) 6-10
- too many server classes (error 922) 6-10

Servers, writing
- context-free with context-sensitive requesters 4-4
- context-sensitive programming 4-13/4-17
- debugging 4-11/4-12
- Guardian 4-2, 4-3
- linkage space considerations 4-3
- nested servers 4-4
- overview 4-1
- Pathway 4-2
- reply formats 4-1
- RSC requester considerations 4-4
- SCREEN COBOL requester considerations 4-3
- sharing by different requesters 4-1
- TMF subsystem
  - application structure 4-5/4-6
  - audited files 4-7
  - fault-tolerant programming 4-10
  - grouping transaction operations 4-8/4-9
  - overview 4-5
  - record locking 4-8
  - restrictions 4-7
  - transaction deadlocks 4-10

Server-class name invalid (error 900) 6-2

Server-to-server communication
- See Nested servers

Single-threading in server design 2-20, 2-23

Software development tools 1-14/1-15

Special registers, PROGRAM-STATUS 4-10

SPI (Subsystem Programmatic Interface)
- description 1-4
- use in managing GDSX processes 2-18
- use in managing servers 1-8, 2-19

Standardizing and testing code 1-3

Static links 3-4

Subsystem Control Facility (SCF), use in managing GDSX processes 2-18

Subsystem Programmatic Interface (SPI)
- See SPI (Subsystem Programmatic Interface)

Synchronization IDs 3-6

Syntax
- See individual procedures

System /T clients, NonStop TUXEDO, interoperating with 4-17

Systems
- expanding 1-7
- NonStop 1-5/1-7
- security 1-6

T

Tables, NonStop SQL/MP 2-10

TAL
- example Pathsend requester program B-1/B-52
- for Pathsend requesters 1-10
- for Pathway servers 1-8
- invoking Pathsend procedures 5-5

Tandem
- Alliance 1-4
- computing fundamentals 1-3/1-7

TCP (terminal control process)
- fault-tolerance role 1-6
- features provided by 1-10
- IDS requesters 2-12
- RSC requesters 2-15

TDP (Transaction Delivery Process) 2-15

TEDIT text editor 1-14

Terminals 2-12

TERMINATION-STATUS and -SUBSTATUS 4-11/4-12

Testing and standardizing code 1-4

Third-party vendors 1-4

Throughput, system 1-6

timelimit parameter, AWAITIOX procedure 5-28
TIMEOUT attribute 5-27
Timeout error (error 40) 4-11/4-12
Timeout errors 4-11/4-12
Timeouts
  invalid value (error 919) 6-9
  Pathsend usage considerations 2-14, 5-27/5-28
  send completion, specifying maximum 5-9, 5-15, 5-19
  servers, specifying 5-27
  server-classes, specifying 5-27
TMF OFF server parameter 3-5, 5-24
TMF (Transaction Management Facility)
  audit-trail files 2-8
  defining transactions 2-6
  description 1-4
  fault-tolerance role 1-6
Too many dialogs (error 927) 6-12
Too many Pathmons (error 921) 6-9
Too many requesters (error 925) 6-11
Too many send requests (error 924) 6-11
Too many server classes (error 922) 6-10
Too many server links (error 923) 6-10
Tools, software development 1-14/1-15
tpinit() ATMI function 3-11
Transaction Application Language
  See TAL
Transaction identifiers and LINKMON processes 5-24
Transaction Management Facility
  See TMF
Transaction processing
  See OLTP
Transaction processing scenario 1-15/1-17
Transactions
  aborted (error 934) 6-14
  aborting 2-8, 2-23
  backout 2-8
Transactions (continued)
  concurrency control 2-7/2-8
  cost per 1-6
  database consistency and concurrency 4-5
  deadlocks 4-10
  defining for the TMF subsystem 2-6
  designing an application with 2-1/2-8
  example abort B-15
  grouping operations 4-8/4-9
  identifier 2-6
  identifier changed (error 930) 6-13
  identifying components 2-4/2-5
  integrity 1-5
  protecting 2-6/2-8
  start with TMF, example B-13
TSCODE, GDSX process 2-16
TUX2PWY translation server 1-13, 4-17
TUXEDO
  See NonStop TUXEDO
TUXEDO requesters (clients)
  writing Pathway servers for 1-13, 4-17
TUXEDO to Pathway translation server 1-13, 4-17

U

Unknown server class (error 914) 6-7
Unsolicited message processing (UMP) 2-12
Unstructured files 2-10
USCODE, GDSX process 2-16
Utilities, software development 1-14/1-15
Utilization of resources 3-8

V

Vendors, third-party 1-4

W

Waited send operations
errors for 6-1
procedure calls for 5-7, 5-13, 5-17
specifying 5-9, 5-15, 5-20
usage considerations 5-23, 5-27
Workstation clients, NonStop TUXEDO, interoperating with 4-17

Special Character

$RECEIVE messages
allocating space 4-3
determining new dialogs 4-13